ITpreneurs™
Effective Learning Solutions

ISTQB® Foundation Extension Agile Tester

Participant Handbook

Table of Contents

ISTQB Foundation Level Extension Agile Tester
Certificate Course
Timetable

Start Time	Duration	Day 1 of 2
08:30	0:15	Welcome and Introductions
		1. Agile Software Development
08:45	0:05	1.1.1 Agile Software Development and the Agile Manifesto
08:50	0:05	**Exercise: 1_1 Values and Principles**
08:55	0:15	1.1.2 Whole-Team Approach
09:10	0:15	1.1.3 Early and Frequent Feedback
09:25	0:10	1.2.1 Agile Software Development Approaches
09:35	0:10	Break
09:45	0:15	1.2.2 Collaborative User Story Creation
10:00	0:30	**Exercise: 1_2 User Stories**
10:30	0:10	1.2.3 Retrospectives
10:40	0:15	1.2.4 Continuous Integration
10:55	0:10	1.2.5 Release and Iteration Planning
11:05	0:10	Break
11:15	0:30	**Practice Exam 1**
11:45	0:10	**Practice Exam 1 Review**
		2. Fundamental Agile Testing Principles, Practices, and Processes
11:55	0:15	2.2.1 Testing and Development Activities
12:10	0:10	2.1.2 Project Work Products
12:20	0:05	2.1.3 Test Levels
12:25	1:00	Lunch
13:25	0:05	2.1.4 Testing and Configuration Management
13:30	0:05	2.1.5 Organizational Options for Independent Testing
13:35	0:10	2.2.1 Communicating Test Status, Progress, and Product Quality
13:45	0:10	2.2.2 Managing Regression Risk with Evolving Manual and Automated Test Cases
13:55	0:10	2.3.1 Agile Tester Skills
14:05	0:10	2.3.2 The Role of a Tester in an Agile Team
		3. Agile Testing Methods, Techniques, and Tools
14:15	0:10	Test-Driven Development, Acceptance Test-Driven Development, and Behavior-Driven Development
14:25	0:10	Break
14:35	0:10	The Test Pyramid
14:45	0:15	Testing Quadrants, Test Levels, and Testing Types
15:00	0:20	The Role of a Tester
15:20	0:35	**Exercise: 3_1 Tester in Scrum Team**
15:55	0:10	Break
16:05	0:15	Assessing Quality Risks in Agile Projects
16:20	0:35	**Exercise: 3_2 Assess Quality Risks**
16:55		Close
	1:00	**Homework: Practice Exams 2 and 3A**

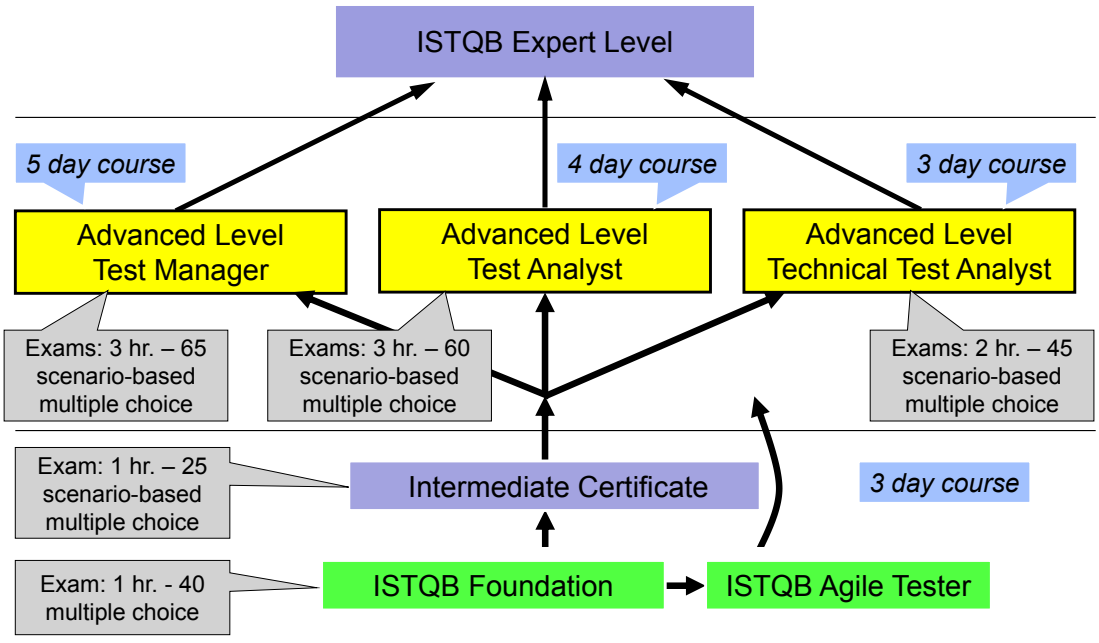
ISTQB Foundation Level Extension Agile Tester
Certificate Course
Timetable

Start Time	Duration	Day 2 of 2
08:30	0:10	**Practice Exams 2 and 3A Review**
08:40	0:15	Estimating Testing Effort Based on Content and Risk
08:55	0:35	**Exercise: 3_3 Estimate Testing Effort**
09:30	0:15	Acceptance Criteria, Adequate Coverage, and Other Information for Testing
09:45	0:10	Break
09:55	0:35	**Exercise: 3_4 Interpret Information**
10:30	0:15	Applying Acceptance Test-Driven Development
10:45	0:35	**Exercise: 3_5 ATDD Test Cases**
11:20	0:10	Break
11:30	0:10	Functional and Non-Functional Black Box Test Design
11:40	0:35	**Exercise: 3_6 Black Box Test Cases**
12:15	0:15	Exploratory Testing and Agile Testing
12:30	1:00	Lunch
13:30	0:35	**Exercise: 3_7 Exploratory Testing**
14:05	0:20	Tools in Agile Projects
14:25	0:40	**Practice Exam 3B**
15:05	0:15	**Practice Exam 3B Review**
15:20	0:30	Break
15:50	1:00	**Exam**
16:50		Close

ISTQB Foundation Extension Agile Tester

Contents and Introduction

ISTQB Professional Certificates



2

Learning objectives (LO)

- the syllabus requires three levels of knowledge:
  - K1: remember, recall (e.g. recognise a definition)
  - K2: understand, give reasons for (e.g. why should testing start early in the life cycle)
  - K3: apply, do, perform (e.g. apply boundary value analysis technique to identify valid boundaries)

3

Agile Tester course content

1. Agile Software Development
2. Fundamental Agile Testing Principles, Practices and Processes
3. Agile Testing Methods, Techniques and Tools

ISTQB Exam

4

Course notes

1. Agile Software Development
2. Fundamental Agile Testing Principles, Practices and Processes
3. Agile Testing Methods, Techniques and Tools
4. References
5. Glossary
6. Syllabus

5

About you …

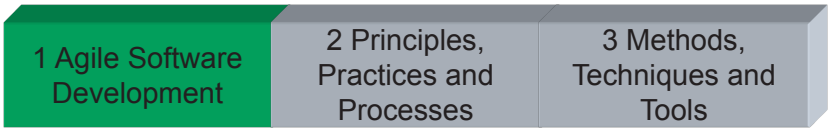
- name/company/job/what you do
- how did you get into testing?
- background/experience
  - testing
  - Agile experience
  - programming/test tools
- why are you here?
  - your objectives (besides passing the exam!)
  - questions/problems in testing

6

ISTQB Foundation Extension Agile Tester

Module 1:
Agile Software Development

Agile Software Development



Contents

Introduction

- Agile development and testing isn't a standard approach
  - ISTQB view
    - ► as defined by collective wisdom мудрост of experts from around the world
- aligns to a set of values and principles
  - Agile manifesto
- many different implementations
  - common methods
    - ► e.g. Scrum, XP, Kanban, etc.

4

Contents

5

The Agile Manifesto

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

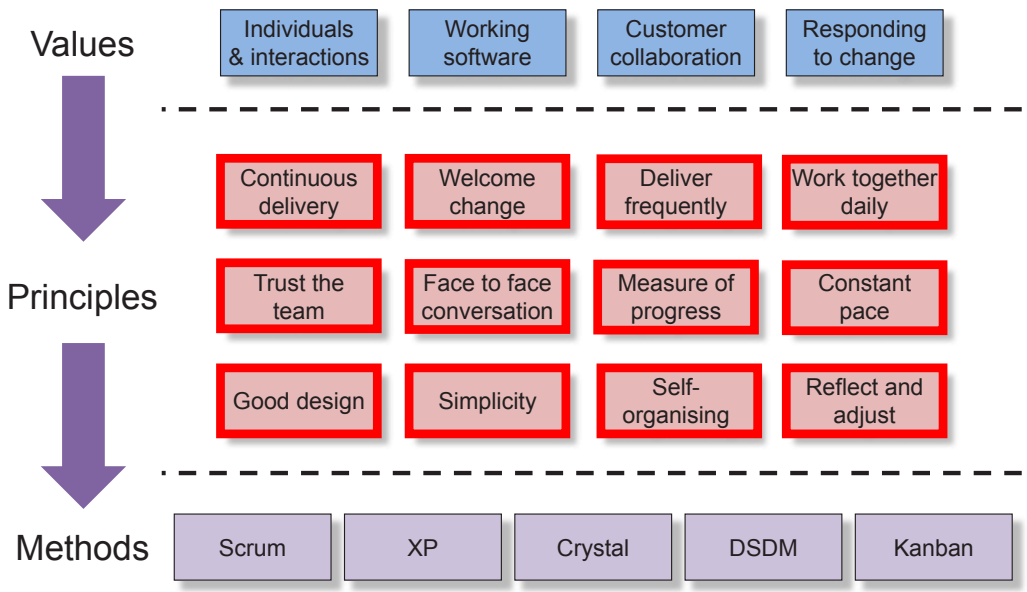
Individuals and interactions	over	processes and tools
working software	over	comprehensiv сеопфатна documentation
customer collaboration	over	contract negotiation
responding to change	over	following a plan

That is, while there is value in the items on the right, we value the items on the left more."

Source: www.agilemanifesto.org

6

The 12 Agile principles



7

Contents

Whole-team approach

- Agile team includes all the skills and knowledge necessary to ensure project success
  - business representatives, programmers*, testers
  - small in size (typically 3 to 9 people)
- ideally co-located
  - facilitates communication and interaction
- daily stand-up meetings
  - involve all team members
  - highlight progress and obstacles to progress
  - short (e.g. 15 minutes)

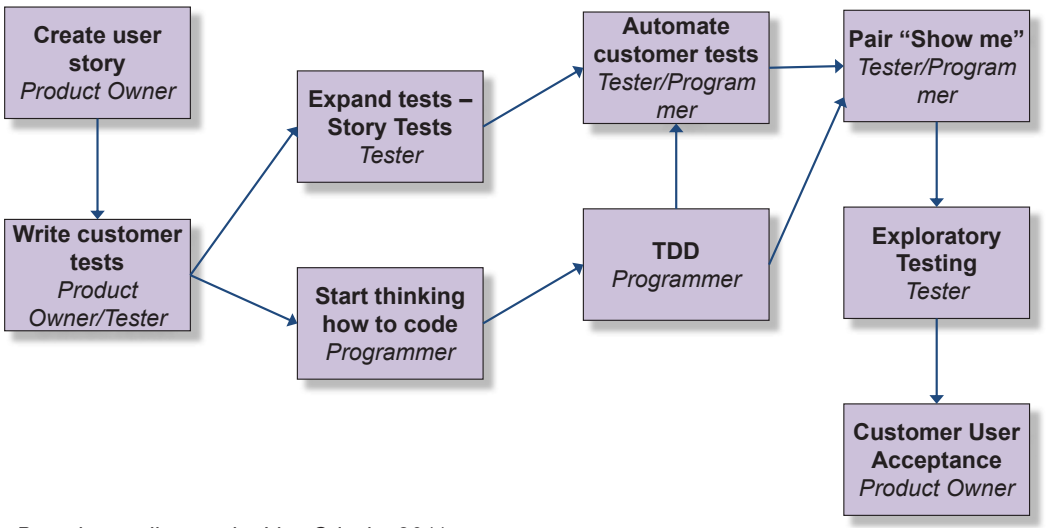
** 'programmer' synonymous with 'developer'*

Roles and activities

- programmers
  - determine how to code
  - test-driven development (TDD)
  - automate tests
- testers
  - expand tests
  - automate tests
  - exploratory testing
- customer/business representatives
  - create user stories
  - write customer tests
  - acceptance testing

10

A possible workflow работниот тек



Based on a diagram by Lisa Crispin, 2011

11

The Power of Three

- a reference to the presence of
  - testers
  - programmers
  - business representatives
- in all feature discussions

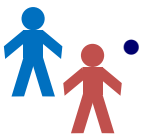
12

Benefits of the whole-team approach

- enhancing подобрување communication
  - daily stand-up meeting
  - frequent conversations
  - frequent meetings

- making quality everyone's responsibility
  - primary focus on quality
  - test lead да доведе objective assessment

- enhancing collaboration
  - co-location
  - pair working



- promoting more effective and efficient team dynamics
  - sharing a common goal
  - working together better

- leveraging skills
  - mixed skills
  - skills transfer

13

Responsibility for quality

- everyone on team focused on
  - ► production of high (desired) quality software
  - ► timed to give maximum value to business
- testers, developers* and business representatives work together
  - ► to ensure the (desired) quality levels are achieved
- tests drive the work
  - ► help the team learn how the application should work
  - ► acceptance criteria
    - – telling us when a task is complete

** 'developer' synonymous with 'programmer'*

The whole team is responsible for quality

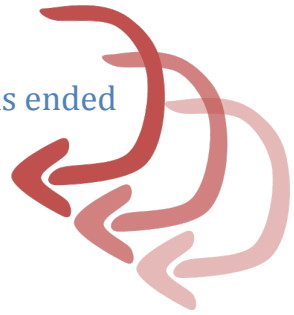
14

Contents

15

Early and frequent feedback

- sequential development
  - customer feedback comes after the project has ended
- Agile development
  - performed in short iterations
  - feedback possible after each iteration
- feedback includes
  - what works well
  - problems
  - what needs changing
  - revised priorities



16

Benefits of early and frequent feedback

- avoid requirement misunderstandings
  - deliver correct software earlier and with less effort
- clarify customer feature requests
  - product better matches customer wishes
- resolving quality problems early
  - fewer problems later, less maintenance
- team productivity assessment
  - frequent comparison of effort estimates with actuals
- project momentum
  - consistent and sustainable pace

17

Contents

18

Common Agile approaches and practices

- Agile approaches/methods
  - Extreme Programming (XP)
  - Scrum
  - Kanban
- practices used within Agile development
  - collaborative user story creation
  - retrospectives
  - continuous integration
  - release and iteration planning

These practices are not unique to Agile development but they are essential to successful Agile development

19

Contents

20

Extreme Programming (XP)

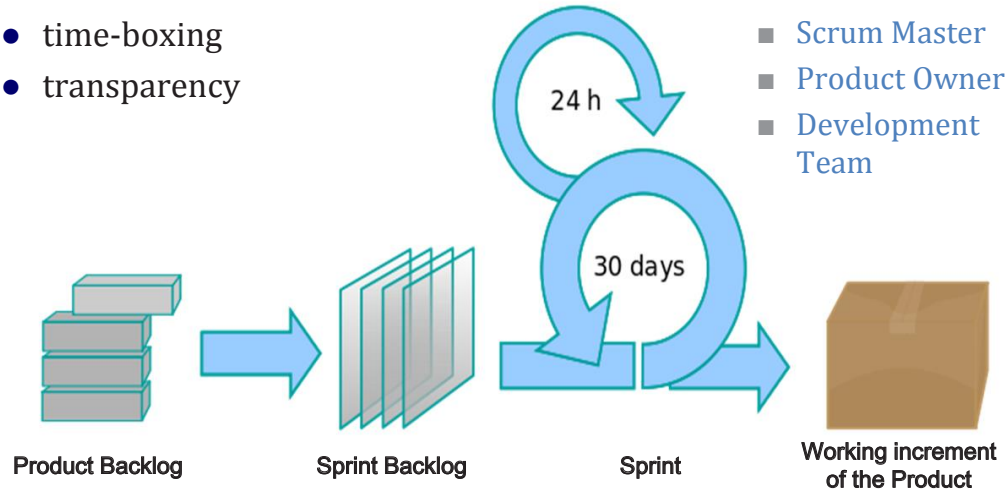
- XP precedes претходиAgile manifesto
  - Agile approach to software development
- Values
  - communication
  - simplicity
  - feedback
  - courage
  - respect
- Principles
  - humanity, economics, mutual взаемно benefit, self-similarity, etc.
- Primary practices
  - ▶ sit together
  - ▶ whole team
  - ▶ informative workspace
  - ▶ energised work
  - ▶ pair programming
  - ▶ stories
  - ▶ weekly cycle
  - ▶ quarterly cycle
  - ▶ slack
  - ▶ ten-minute build
  - ▶ continuous integration
  - ▶ test first programming
  - ▶ incremental постепено design

21

Scrum

- definition of done
- time-boxing
- transparency

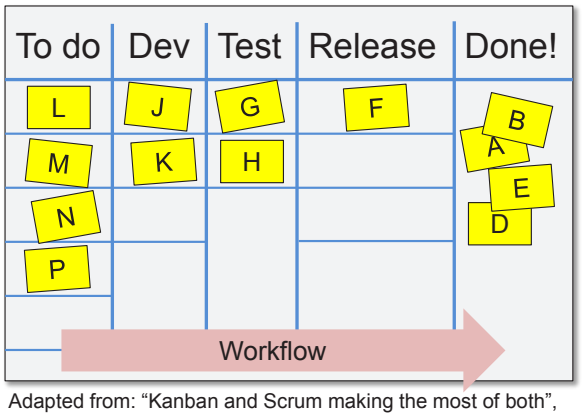
- Roles:
  - Scrum Master
  - Product Owner
  - Development Team



22

Kanban

- Kanban board
  - shows the 'value chain'
  - columns: related activities
- work-in-progress limit
  - max number activities
- optimise lead time
  - minimise average time for task to move across board
- time-boxing and iterations are optional
  - release deliverables item by item



Adapted from: "Kanban and Scrum making the most of both", Henrik Kniberg & Mattias Skarin

23

Contents

The requirements challenge

- traditional requirement specifications
  - contribute to project failure as much as project success
  - "63% of bugs traceable back to the requirements specification"

Source: IBM

- typical problems
  - incomplete
  - lack detail
  - ambiguous
  - contradictory
  - wrong
- typical causes
  - miscommunication between users, developers and testers
  - no common understanding
  - users not fully understanding their real needs

User Stories: the Agile solution

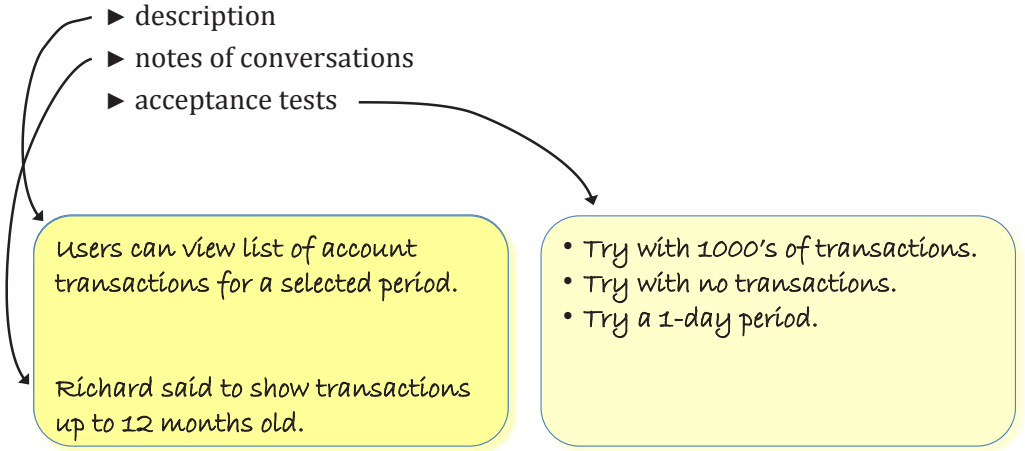
- requirements captured in form of written 'user stories'
  - short, simple description of a feature характеристика
  - from perspective of end user
- written by user/customer/business representative
  - testers and developers may also contribute
- shared vision
  - achieved by frequent informal reviews while user stories are being developed
- not complete
  - starting point for on-going discussions
  - leading to more information
    - ► notes (of conversations)
    - ► acceptance criteria (tests)

Story + Tests + Conversation = Requirement

26

User story example

- a user story
  - three parts (usually hand-written):
    - ► description
    - ► notes of conversations
    - ► acceptance tests



27

User story notes and acceptance criteria

Notes

- serve as a reminder of conversations
  - not detailed documentation
- notes are open questions
  - answered during subsequent conversations

Acceptance criteria

- reflect customer expectations
  - functional and non-functional
- provide more detail about the feature
  - in the form of tests
- tells the team when a user story has been implemented
  - when all tests pass

28

INVEST: characteristics of a good user story

- **I**ndependent
  - stories are easier to work with when they are independent
- **N**egotiable
  - important details gained through discussion during implementation
- **V**aluable
  - to the customer (either user or purchaser)
- **E**stimatable
  - the story is not too big
  - sufficient domain and technical knowledge exists
- **S**mall
  - but not too small
  - and not too big
- **T**estable
  - and automatable
  - non-functional stories most likely to be untestable

29

The 3C concept

CARD

Users can view list of account transactions for a selected period.

Richard said to show transactions up to 12 months old.
What if user has multiple accounts?
Output data privacy warning?

User story

Notes

Questions

CONVERSATION

- Try with 1000's of transactions.
- Try with no transactions.
- Try a 1-day period.
- Try period of exactly 12 months
- Try period of 12 months + 1 day

CONFIRMATION

Tests

30

User story templates

- template guides the structure and content of a user story
  - e.g.

    As a <role>,
    I want <function>,
    so that <business value>
  - e.g.

    As a smart phone user,
    I want the app to track my position throughout the day,
    so that I can see how far I have travelled.
- benefits
  - ensures important information not omitted изоставени-исфрлени
    - ► e.g. <role> or <business value>
  - helps ensure consistency
    - ► everyone knows what to expect
    - ► easier to understand and review
- variety of templates
  - also acceptance criteria templates

31

Contents

32

Retrospective meetings

- whole team meets at end of iteration
  - highlight successes
  - identify improvement opportunities
    - ► i.e. things that could be done better
  - also identify activities that
    - ► should be started
      - – because it may help us do better in the next iteration
    - ► should be continued
      - – because it worked well in this iteration
    - ► should be stopped
      - – because it didn't work well in this iteration

33

Topics and conduct

Topics

- anything relevant to iteration
  - processes, people, tools, organisation, relationships

Conduct

- professional environment
  - mutual trust and respect
- all comments
  - objective, constructive
- team members
  - must feel safe
    - ► will not be criticised

Remember it is a team effort

34

Being selective

- though many improvement ideas may be identified
  - should implement only a few (1 or 2?) in next iteration
    - ► more likely to be successful
    - ► improvement effort will not detract from delivering value
  - small but frequent improvements
    - ► help achieve continuous improvement at sustained pace
- how to select?
  - whole team prioritises
    - ► e.g. give every member 10 points to vote with
      - – all point on one idea or spread across a number of ideas

35

Test improvement

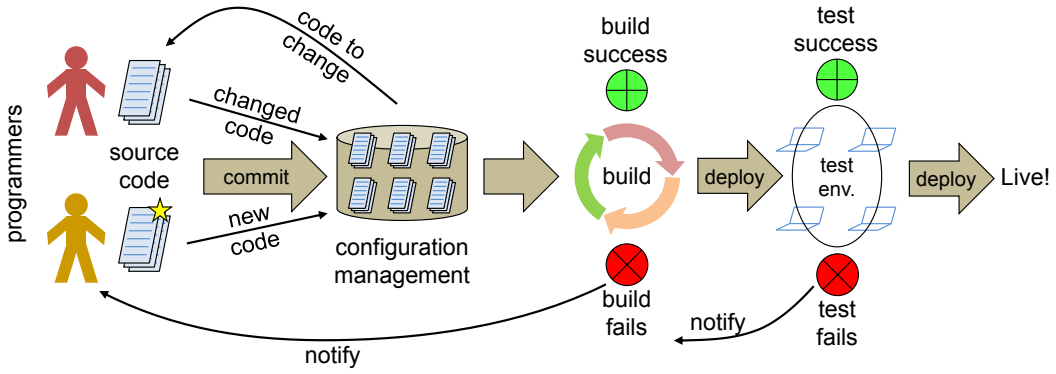
- all team members
  - can identify test successes and improvements
- topics include
  - test effectiveness
  - test productivity
  - test case quality
  - team satisfaction
  - testability of
    - ► applications, user stories, features and system interface

36

Contents

37

Steps to deliver a software product



this is integrating all the parts of a product …
to make it available for testing and production

38

Effect of build frequency

Infrequent builds

- accumulates defects
  - more regression risk
- longer time to integration and system testing
  - longer to feedback to programmers
  - programmers working on a different feature
  - more difficult to isolate and fix defects
- poorer quality software

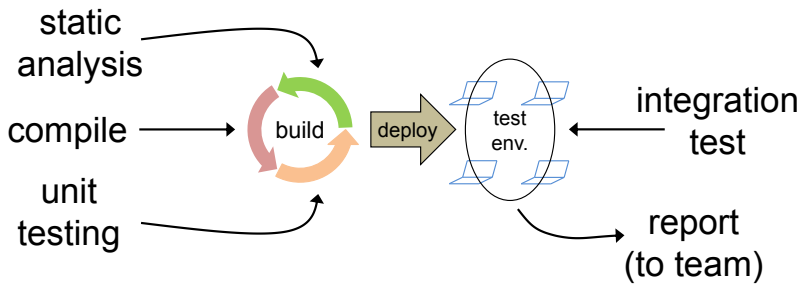
Frequent builds

- fewer defects
  - less regression risk
- shorter time to integration and system testing
  - quicker to feedback to programmers
  - programmers still engaged on feature
  - easier to isolate and fix defects
- better quality software

key benefit

39

Continuous integration: automating the build process

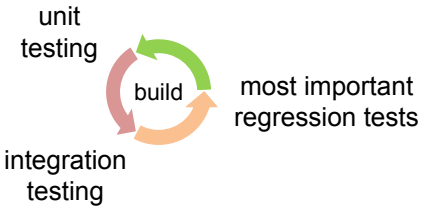


- build process:
  - static analysis
  - compile
  - unit test
  - deploy
  - integration test
  - report
- frequency: at least daily
- additional checks and testing
  - detect defects early
- elapsed time ideally <10 mins.
  - if it breaks, only 10 mins. wait after fix
- most important tests first

40

Automated testing as part of the build process

- within the build process
  - component testing
  - integration testing
    - ► minimum requirement is to cover integration risks
  - other automated tests
    - ► e.g. most important regression tests
    - ► if they don't make the elapsed time for the build process too long
- purpose
  - find defects early
  - feedback on code quality
- defects found at any stage of the build process stop the build
  - until issues are resolved
  - test results visible to all team members



41

Automated testing separate from the build process

- additional automated testing
  - includes regression tests not run as part of build process
  - regression tests to cover as much functionality as possible
    - ► including user stories of earlier iterations
  - free testers to focus manual testing on
    - ► new features
    - ► changes
    - ► defect fixes
- new executables available more frequently
  - therefore can benefit from more frequent automated test runs
    - ► not necessarily on every build e.g. nightly

42

Build tools

- additional tasks initiated by build tools
  - other automated tests
  - measure and profile performance
  - extract (from code) and format documentation
  - facilitate manual QA
- aim to improve quality and reduce delivery time

- can be linked to automatic deployment tools
  - deploys build into one or more environments
    - ► develop
    - ► test
    - ► staging
    - ► production
- reduces errors and avoids delays

43

Benefits and risks of continuous integration

Benefits

- earlier defect detection
- regular/quick feedback on code quality and decisions taken
- up-to-date software version
- reduced regression risk
- increased confidence
- visible progress
- reduces schedule risks
- constant availability of s/w

Risks

- tool introduction and maintenance
- CI process implementation
- additional resources for test automation
- good test coverage essential
- over-reliance on component tests

44

Contents

45

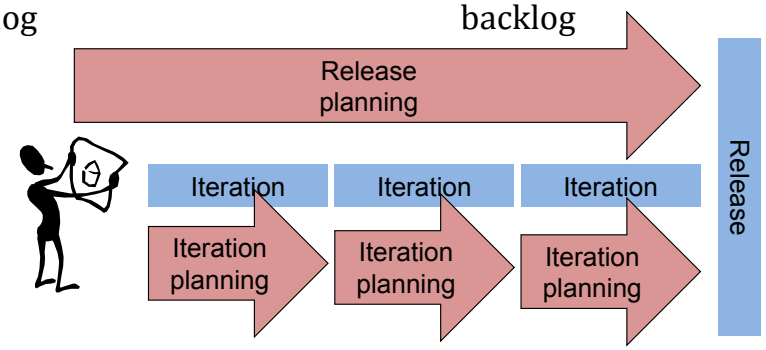
Release and iteration planning

Release planning

- occurs at start of project (and on-going)
  - scope is whole project
- concerned with product backlog

Iteration planning

- occurs at start of each iteration (and on-going)
  - scope is single iteration
- concerned with iteration backlog



46

Release planning

- meetings involve the whole team
  - gain understanding of customers' vision
  - high-level view of what will be in the release
    - ► estimate number of iterations
- business representatives
  - establish and prioritise user stories
  - large stories (epics) split into smaller stories
- based on the stories, the team
  - estimates effort (high-level)
  - identifies
    - ► risks
    - ► dependencies between stories
    - ► other story ordering factors

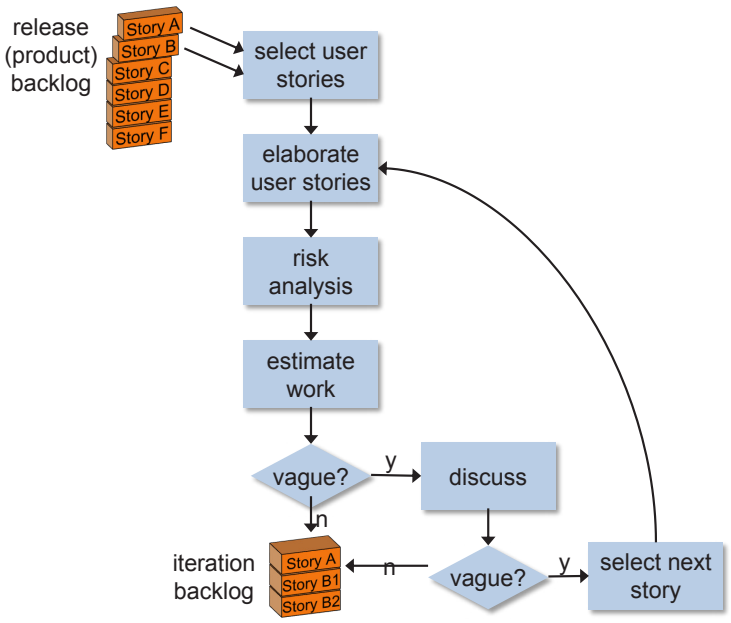
47

Testers involvement in release planning

- consider impact of planned features on the existing system
- clarify assumptions
- identify resources for testing
  - skills
  - test environments
  - tools, etc.
- adding value
  - ensure testable
    - ► user stories
    - ► acceptance criteria
  - project and quality risk analysis
  - estimating test effort
  - defining test levels
  - test planning for release

48

Iteration planning



49

Selecting stories

- stories estimated in 'story points' (or other units)
- team velocity is number of story points (or other units) implemented in previous iterations

E.g. if team velocity = 13

Release Backlog	Points
Story A	3
Story B	5
Story C	2
Story D	13
Story E	3
Story F	8

EITHER

	Iteration Backlog	Points
Iteration 1	A,B,C,E	13
Iteration 2	D	13
Iteration 3	F	8

OR

Release Backlog	Points
Story D1	3
Story D2	8
Story D3	2
::::::::::::	:::

THIS

	Iteration Backlog	Points
Iteration 1	A,B,C,D1	13
Iteration 2	D2,D3,E	13
Iteration 3	F	8

50

Testers involvement in iteration planning

similar to release planning

- adding value:
  - ensure testable
    - ► user stories
    - ► acceptance criteria
  - project/quality risk analysis

specific to iteration planning

- adding value:
  - create acceptance tests
  - breakdown stories into tasks
    - ► particularly testing tasks
  - identify functional and non-functional aspects to be tested
  - estimate test effort
    - ► for tasks
  - test automation

51

Changes to release plans

- release plans may change: triggered by
  - internal factors
    - ► delivery capabilities
    - ► velocity
    - ► technical issues
  - external factors
    - ► new markets and opportunities
    - ► new competitors
    - ► business threats
      - – changing release objective or target dates
- changes can challenge testers
  - altering priorities part way through a release or iteration
    - ► affects ordering of tests
    - ► and types of testing
  - when test basis changes
    - ► so too must the tests
  - can waste early effort on test design

52

Test planning

- part of both release and iteration planning
- specific issues include:
  - scope of testing
  - assignment of test activities
  - test environment and data
  - timing and dependencies
    - ► when to run tests
    - ► sequencing of tests
    - ► test prerequisites
    - ► other dependencies
  - project and quality risks

53

Summary

- Agile Manifesto values
  - individuals/interactions, working software, customer collaboration, responding to change
- key features of Agile
  - whole-team approach
  - short iterations: early and frequent feedback
- common methods
  - Extreme Programming (XP)
  - Scrum
  - Kanban
- essential practices
  - collaborative user story creation
    - ► 3C = card + conversation + confirmation
  - retrospectives
    - ► start, stop, continue
  - continuous integration
    - ► automate daily build process
  - release and iteration planning
    - ► whole-team selects user stories

54

ISTQB Foundation Extension Agile Tester

Session 1:

Agile Software Development

Terms
Agile Manifesto, Agile software development, iteration backlog, product backlog, release backlog, test automation, test basis, test oracle, user story.

From the ISTQB Glossary

agile manifesto: statement on the values that underpin agile software development. The values are:
- individuals and interactions over processes and tools
- working software over comprehensive documentation
- customer collaboration over contract negotiation
- responding to change over following a plan.

agile software development: A group of software development methodologies based on iterative incremental development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams.

test automation: The use of software to perform or support test activities, e.g. test management, test design, test execution and results checking.

test basis: All documents from which the requirements of a component or system can be inferred. The documentation on which the test cases are based. If a document can be amended only by way of formal amendment procedure, then the test basis is called a frozen test basis.

test oracle: A source to determine expected results to compare with the actual result of the software under test. An oracle may be the existing system (for a benchmark), other software, a user manual, or an individual's specialized knowledge, but should not be the code.

user story: A high-level user or business requirement commonly used in agile software development, typically consisting of one or more sentences in the everyday or business language capturing what functionality a user needs, any non-functional criteria, and also includes acceptance criteria.

1.1 The Fundamentals of Agile Software Development

Learning Objectives

FA-1.1.1	K1	Recall the basic concept of Agile software development based on the Agile Manifesto.
FA-1.1.2	K2	Understand the advantages of the whole-team approach.
FA-1.1.3	K2	Understand the benefits of early and frequent feedback.

Like any software development approach, there are many variations to the implementation of Agile development. To be Agile is not to follow a standard process but it is to align with a set of values and practices. In this section we explain the ISTQB view of Agile (as defined by the collective wisdom of many experts from around the world). It is important for all Agile team members to understand these values and principles.

How the Agile values and principles are implemented in practice varies greatly from one organisation to another. However, there are a number of common methods that are used to help achieve alignment to the Agile values and principles and these are described in more detail in the next subsection.

The way testers work on an Agile project differs significantly from to the way they work on traditional projects. In this subsection we also explore the 'whole-team approach' that is central to any Agile approach and explain the benefits of early and frequent feedback from the business to the Agile team.

1.1.1 Agile Software Development and the Agile Manifesto

Learning Objective

FA-1.1.1	K1	Recall the basic concept of Agile software development based on the Agile Manifesto.

In 2001, a group of individuals, representing the most widely used lightweight software development methodologies, agreed on a common set of values and principles that became known as the Manifesto for Agile Software Development or the Agile Manifesto.

The Agile Manifesto contains four statements of values:

individuals and interactions	over	*processes and tools*
working software	over	*comprehensive documentation*
customer collaboration	over	*contract negotiation*
responding to change	over	*following a plan*

The Agile Manifesto argues that although the concepts on the right have value, those on the left have greater value. Let us take a closer look at each of these in turn …

Individuals and Interactions

Agile development is very people-centred. Agile approaches recognise that it is teams of people that build software so the whole approach is oriented toward what makes people work more effectively. Agile approaches do not prescribe a particular process, tool or method but require the team members to continuously communicate and interact with each other and with other stakeholders. This approach has enabled teams to work more effectively than those that are more reliant on tools or processes.

Working Software

From a customer perspective, working software is much more useful and valuable than overly detailed documentation and it provides an opportunity to give the development team rapid feedback. In addition, because working software, albeit with reduced functionality, is available much earlier in the development lifecycle, Agile development can give significant time-to-market advantage.

Agile development is especially useful in rapidly changing business environments where the problems and / or solutions are unclear or where the business wishes to develop innovative new products.

Customer Collaboration

People (including our customers) often find great difficulty in specifying the system that they require. It is practically impossible to fully specify complex systems precisely and in detail at the start of a project. No matter how much time and effort we put into a specification it is not until we see the product and are able to start using it that we properly understand our detailed requirements.

Our customer requirements will evolve throughout the project and it is important that these are communicated to the whole development team. Collaborating directly with the customer improves the likelihood of the development team understanding exactly what the customer requires.

While having contracts with customers may be important, working in regular and close collaboration with them is likely to bring more success to the project.

Responding to Change

Change is inevitable in software projects. The environment in which the business operates, legislation, competitor activity, technology advances and other factors can have major influences on the project and its objectives. For project success these factors should be accommodated by the development process. As such, having flexibility in work practices to embrace change is more important than simply adhering rigidly to a plan.

For more information about the Agile Manifesto and its origins see [Web-2].

Principles

The core Agile Manifesto values are captured in twelve principles:

1. Our highest priority is to satisfy the customer through **early and continuous delivery of valuable software**.
2. **Welcome changing requirements**, even late in development. Agile processes harness change **for the customer's competitive advantage**.
3. **Deliver working software frequently**, at intervals of between a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must **work together daily** throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and **trust them to get the job done**.

6. The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**.
7. **Working software is the primary measure of progress**.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to **maintain a constant pace indefinitely**.
9. Continuous attention to **technical excellence and good design** enhances agility.
10. **Simplicity**—the art of maximising the amount of work not done – **is essential**.
11. The best architectures, requirements, and designs emerge from **self-organising teams**.
12. At regular intervals, **the team reflects** on how to become more effective, then tunes **and adjusts** its behaviour accordingly.

As stated previously there is not a single Agile approach. There are many different methods that each implement some or all of the values and principles described above. In Section 1.2.1 "Agile Software Development Approaches" we describe three of the more common methods.

1.1.2 Whole-Team Approach

Learning Objective

FA-1.1.2 K2 Understand the advantages of the whole-team approach.

One of the most significant differences between Agile development and the more traditional development approaches is the Agile "whole-team" approach. Rather than having a project staffed by members from different departments (e.g. programmers from the development department and testers from the testing department) the Agile project is undertaken by a single multi-disciplinary team. All the skills and knowledge necessary to ensure project success are brought together in one team. (Note that the terms 'programmer' and 'developer' are synonymous in these notes.)

Team size, location and meetings

The team should be relatively small; successful teams have been observed with as few as three people and as many as nine. Ideally, the whole team shares the same workspace, as co-location significantly improves communication and interaction, and this is important to the Agile values and principles.

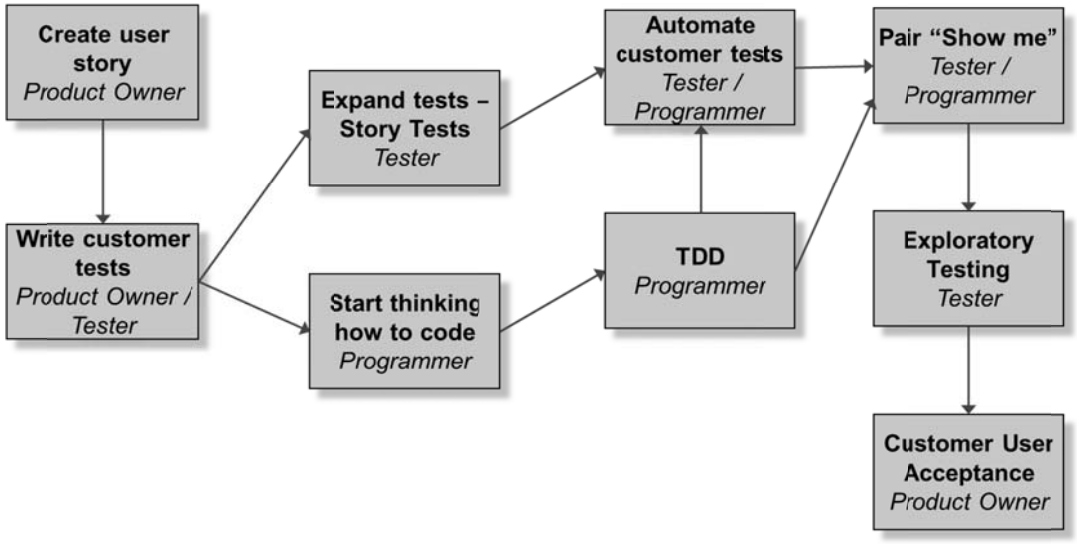
The whole-team approach is supported through the daily stand-up meetings (see Section 2.2.1) involving all members of the team, where work progress is communicated and any impediments to progress are highlighted. These daily stand-up meetings are literally stand-up meetings – the whole team meets standing up. This helps keep the meeting short as team members are less likely to waste time talking about things that are not important.

Roles and activities

Agile principles encourage team members to take on multiple activities such that any team member can take on any type of task. This suggests that Agile teams should avoid having specialised roles and encourage all team members to transfer their skills to others in the team as much as possible. That said, in many cases there will be some distinct roles such as programmers who write the code, software architects who design software solutions and performance and security experts who bring their special knowledge and skills to the project. Testers (with or without programming skills) may help the programmers develop their tests and help them to better understand business-oriented (acceptance) tests. The programmers

may use Test-Driven Development (TDD) a development approach that requires automated tests to be implemented before the code is written. (TDD is described in Section 3.1.1.)

In addition to the programmers and testers an Agile team includes representatives from the customer and/or other business stakeholders. This subset of the Agile team represents the business and, amongst other tasks, determines product features, the priorities and the acceptance criteria for each delivery. They may also create user stories and acceptance tests, and undertake acceptance testing. Testers are likely to help the customer representatives with testing tasks such as specifying acceptance tests.



Based on a diagram by Lisa Crispin, 2011

Power of Three

The whole team is involved in any consultations or meetings in which product features are presented, analysed, or estimated. This concept of involving testers, developers, and business representatives in all feature discussions is known as the power of three [Crispin08].

Benefits of the whole-team approach

The use of a whole-team approach to product development is one of the main advantages of Agile development. Its benefits include:

- enhancing communication and collaboration within the team
- enabling the various skill sets within the team to be leveraged to the benefit of the project
- making quality everyone's responsibility
- promoting more effective and efficient team dynamics

Responsibility for quality

The goal of Agile development is the production of high quality software within time scales that provide maximum value to the business. Everyone on the team is focused on this goal, not just testers or other quality assurance people.

The whole team is responsible for quality in Agile projects.

The essence of the whole-team approach lies in the testers, developers, and the business representatives working together in every step of the development process. While there may need to be specialists in the team, such as expert testers or software architects, any task might be completed by any team member. Testing tasks such as test design, test automation, and exploratory testing are the responsibility of the whole team and are likely to be undertaken by different team members at different times throughout the project.

Tests drive the work. From the component testing level up, tests are used to help the team learn how the application should work and tell us when we're done with a task.

Testers work closely with both developers and business representatives to ensure that the desired quality levels are achieved. This includes supporting and collaborating with business representatives to help them create suitable acceptance tests, working with developers to agree on the testing strategy, and deciding on test automation approaches. In this way testers transfer and extend testing knowledge to other team members and influence the development of the product.

1.1.3 Early and Frequent Feedback

Learning Objective

FA-1.1.3 K2 Understand the benefits of early and frequent feedback.

When sequential development approaches are used, the customer often does not see the product until the project is nearly completed. At that point, it is often too late for the development team to effectively address any issues the customer may have.

Agile projects are undertaken as a series of short (typically 2 week) iterations. Every iteration has the potential to deliver a new version of the product with additional and /or revised features. When the new version is delivered to the customer the project team should receive feedback from the customers about the quality of product, revealing how well the product matches their expectations.

Another approach to delivering new versions of the product to the customer in even shorter and more frequent intervals is called continuous integration. This is described in Section 1.2.4 "Continuous Integration").

Benefits of early and frequent feedback

The benefits of early and frequent feedback include:

- Avoiding requirements misunderstandings

  It is more likely customers will identify issues with the product (even an incomplete product) in front of them than they will when seeing only documented specifications. Issues identified and corrected early are cheaper to fix and feedback also helps the developers better understand the requirements so further work on the product will be more aligned to the customer true requirements.

- Clarifying customer feature requests

  Making customer feature requests available for use early helps customers assess their own requests, steering the development of the project such that it better reflects what the customer wants. The early and frequent feedback helps the team focus on the features with the highest business value, or associated risk, and these are delivered to the customer first. Also, by getting frequent customer feedback as the project progresses, Agile teams can incorporate most new changes into the product development process.

- Resolving quality problems sooner

  Discovering (via continuous integration), isolating, and resolving quality problems early ensure a better quality product from the start of the project and throughout the product's development. With fewer quality problems to detect and correct later on the cost of development and operational maintenance is lower.

- Regular team productivity assessment

  Early and frequent feedback provides information to the Agile team regarding its productivity and ability to deliver. This helps to fulfil the 'reflect and adjust' Agile

principle. For example, the effort estimates for features are compared with the actual effort spent so each subsequent round of estimation can be improved. The team should ask itself “What could help us go faster?” and “What is preventing us from doing so?” in order to identify changes that would help them work faster (to achieve more within an iteration).

- Promoting consistent project momentum.

  Maintaining a constant pace is one of the Agile principles. This is achieved by working in short iterations and finite tasks together with daily reviews of progress and the obstacles to progress.

1.2 Aspects of Agile Approaches

Learning Objectives

FA-1.2.1	K1	Recall Agile software development approaches.
FA-1.2.2	K3	Write testable user stories in collaboration with developers and business representatives.
FA-1.2.3	K2	Understand how retrospectives can be used as a mechanism for process improvement in Agile projects.
FA-1.2.4	K2	Understand the use and purpose of continuous integration.
FA-1.2.5	K1	Know the differences between iteration and release planning, and how a tester adds value in each of these activities.

There are several Agile approaches / methods, each of which implements the values and principles of the Agile Manifesto in different ways. In this section we consider three popular representatives of Agile approaches:

- Extreme Programming (XP)
- Scrum
- Kanban

In addition to the approaches / methods, there are a number of practices used within Agile development. Note that we say “practices used within Agile development” rather than “Agile practices”. They are not unique to Agile approaches but have been used by successful software development projects long before “agile development” became popular. However, they are essential to successful Agile development. Some of the more common practices include the following:

- collaborative user story creation
- retrospectives
- continuous integration
- release and iteration planning

Each of these common practices is described below, but first, we take a look at the three approaches mentioned above.

1.2.1 Agile Software Development Approaches

Learning Objective

FA-1.2.1	K1	Recall Agile software development approaches.

1.2.1.1 Extreme Programming (XP)

Extreme Programming (XP), originally introduced by Kent Beck [Beck04], is an Agile approach to software development described by its own set of values, principles and

practices. These are consistent with the Agile values, principles and practices described earlier.

Values

There are five values that XP embraces to guide development:

- communication

  Everyone is part of the team and we communicate face to face daily. We will work together on everything from requirements to code.

- simplicity

  We will do what is needed and asked for, but no more. This will maximise the value created for the investment made to date.

- feedback

  We will take every iteration commitment seriously by delivering working software. We demonstrate our software early and often then listen carefully and make any changes needed. We will talk about the project and adapt our process to it, not the other way around.

- courage

  We will tell the truth about progress and estimates. We don't document excuses for failure because we plan to succeed. We don't fear anything because no one ever works alone. We will adapt to changes whenever they happen.

- respect

  Everyone gives and feels the respect they deserve as a valued team member. Everyone contributes value even if it's simply enthusiasm.

Principles

XP describes fourteen principles as additional guidelines:

- humanity
- economics
- mutual benefit
- self-similarity
- improvement
- diversity
- reflection
- flow
- opportunity
- redundancy
- failure
- quality
- baby steps
- accepted responsibility

Primary practices

XP describes thirteen primary practices:

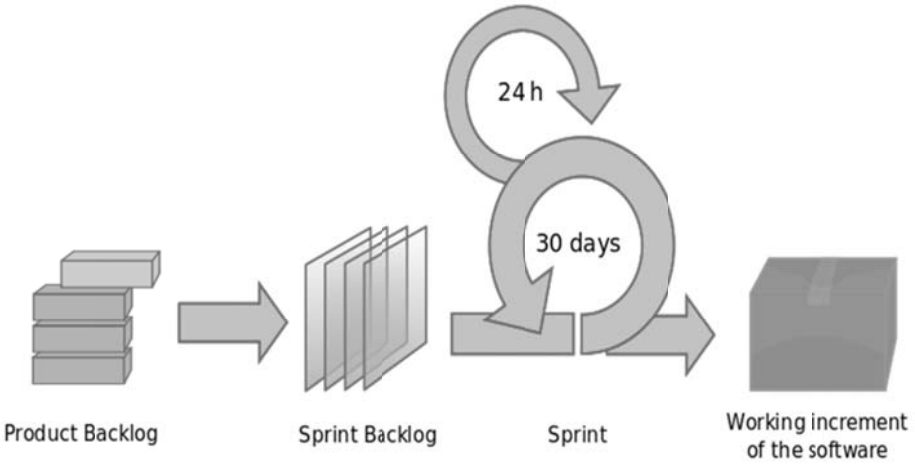
- **Sit together:** develop in an open space big enough for the whole team.
- **Whole team:** a cross-functional team that sits together as members of one team.
- **Informative workspace:** a workspace that meets human needs, and a place for big visible charts.
- **Energised work:** a reinterpretation of "40-hour week" and "sustainable pace" – work only as many hours as you can be productive and only as many hours as you can sustain.
- **Pair programming:** write all production programs with two people
- **Weekly cycle:** an iteration: plan, write tests & code.
- **Quarterly cycle:** plan work a quarter at a time using themes.
- **Slack:** include minor tasks in the plan that can be dropped if you get behind.
- **Ten-minute build:** automatically build and test everything in ten minutes.
- **Continuous integration:** integrate and test changes after no more than a couple of hours.
- **Test first programming:** write a failing automated test before writing any code.

sitting at one machine.
- **Stories:** units of customer-visible functionality, best done as real cards on a real wall.
- **Incremental design:** invest in the design of the system every day.

Many of the Agile software development approaches in use today are influenced by XP and its values and principles. For example, Agile teams following Scrum often incorporate XP practices.

See [Web-3] and [Web-4] for more information on XP.

1.2.1.2 Scrum



Scrum is an Agile management framework that contains the following constituent instruments and practices [Schwaber01]:

- Sprint: fixed length iteration.

  Scrum divides a project into iterations (called sprints) of fixed length (usually two to four weeks).

- Product Increment: potential release, result of a sprint.

  Each sprint results in a potentially releasable / shippable product (called an increment).

- Product Backlog: prioritised, planned product items.

  The product owner manages a prioritised list of planned product items (called the product backlog). The product backlog evolves from sprint to sprint (called backlog refinement).

- Sprint Backlog: items to be done in the sprint.

  At the start of each sprint, the Scrum team selects a set of highest priority items (called the sprint backlog) from the product backlog. Since the Scrum team, not the product owner, selects the items to be realised within the sprint, the selection is referred to as being on the pull principle rather than the push principle.

- Definition of Done: completion criteria for a sprint or test level or story / feature etc.

  To make sure that there is a potentially releasable product at each sprint's end, the Scrum team discusses and defines appropriate criteria for sprint completion. The discussion deepens the team's understanding of the backlog items and the product requirements.

- Time-boxing: putting strict (maximum) time boundaries around an action or activity.

Only those tasks, requirements, or features that the team expects to finish within the sprint are part of the sprint backlog. If the development team cannot finish a task within a sprint the associated product feature is removed from the sprint and the task is moved back into the product backlog. Time-boxing applies not only to tasks, but in other situations (e.g. enforcing meeting start and end times).

- Transparency: keep everything about a project visible to everyone.

  The development team reports and updates sprint status on a daily basis at a meeting called the daily scrum. This makes the content and progress of the current sprint, including test results, visible to the team, management and all interested parties. For example, the development team can show sprint status on a whiteboard.

Scrum defines three roles

- Scrum Master

  Ensures that Scrum practices and rules are implemented and followed, resolves any violations, resource issues, or other impediments that could prevent the team from following the practices and rules. This person is not the team lead, but a coach.

- Product Owner

  Represents the customer, and generates, maintains, and prioritises the product backlog. This person is not the team lead.

- Development Team

  Develop and test the product. The team is self-organised: There is no team lead, so the team makes the decisions. The team is also cross-functional (see Sections 2.3.2 and 3.1.4).

Scrum (as opposed to XP) does not dictate specific software development techniques (e.g. test first programming). Furthermore, Scrum does not provide guidance on how testing has to be done in a Scrum project.

For more information on Scrum see [Web-5] and [Web-6].

1.2.1.3 Kanban

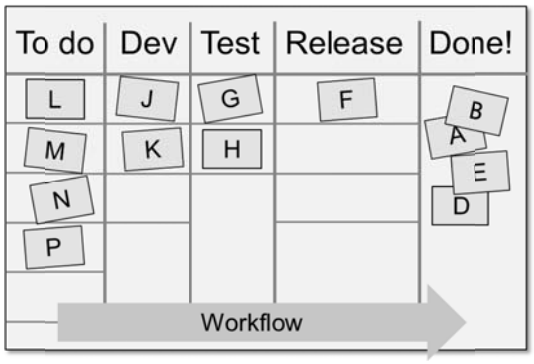
Kanban [Anderson13] is a management approach that is sometimes used in Agile projects. The general objective is to visualise and optimise the flow of work within a value-added chain.

Kanban uses three instruments [Linz14]:

- Kanban Board:

  The value chain to be managed is visualised by a Kanban board. Each column shows a station, which is a set of related activities, e.g., development or testing. The items to be produced or tasks to be processed are symbolised by tickets. The progress of the tasks is indicated by the tickets moving from left to right across the board through the stations.

A Simple Kanban Board



Adapted from: “Kanban and Scrum making the most of both”,
Henrik Kniberg & Mattias Skarin.

For a more detailed example of a Kanban Board see [Web-7].

- Work-in-Progress Limit:

  The amount of parallel active tasks is strictly limited. This is controlled by the maximum number of tickets allowed for a station and / or globally for the board. Whenever a station has free capacity, the worker pulls a ticket from the predecessor station.

- Lead Time:

  Kanban is used to optimise the continuous flow of tasks by minimising the (average) lead time for the complete value stream.

Kanban features some similarities to Scrum. In both frameworks, visualising the active tasks (e.g. on a public whiteboard) provides transparency of content and progress of tasks. Tasks not yet scheduled are waiting in a backlog and moved onto the Kanban board as soon as there is new space (production capacity) available.

Iterations or sprints are optional in Kanban. The Kanban process allows releasing its deliverables item by item, rather than as part of a release. Time-boxing may be used as a synchronising mechanism, but is optional, unlike in Scrum, which synchronises all tasks within a sprint.

For more information on Kanban see [Web-8] and [Web-12].

1.2.2 Collaborative User Story Creation

Learning Objective		
FA-1.2.2	K3	Write testable user stories in collaboration with developers and business representatives.

The requirements challenge

Poor requirement specifications are a common contributor to project failure. Not having a clear understanding of the business requirements can lead to redundant, incomplete or contradictory features being implemented. In such situations the final system delivery may not provide the end users with what they need in order for them to carry out their tasks with the necessary effectiveness and efficiency.

Specification problems typically result from miscommunication between the users, the developers and the testers. Mike Cohn [Cohn04] states “Software requirements is a communication problem. Those who want the new software (either to use or to sell) must communicate with those who will build the new software”. When this communication breaks down there may be no common understanding or shared vision of what the system is meant to do. Furthermore, the users may not fully understand their real needs or at least they may overlook important details.

User Stories

In Agile development, requirements are captured in part in the form of written user stories. Each user story is a short and simple description of a feature told from the perspective of the person who desires the new capability. These are usually written by a business representative, user or customer of the system but testers and developers may also contribute user stories.

In sequential development, the shared vision of a feature is accomplished through formal reviews after requirements are written; in Agile development, the shared vision is accomplished through frequent informal reviews while the requirements are being developed.

The user stories do not provide all the necessary information. Rather, they provide a starting a point for on-going conversations between all the Agile team members: business representatives, testers and developers. Rachael Davies says user stories *represent* user requirements rather than *document* them [Davies01].

Through the conversations, Agile teams collate more information about each user story in the form of notes and tests (acceptance criteria) that together with the written user story serve to give the whole team a better understanding or shared vision of the required feature. The notes serve as a reminder of the conversations rather than documenting them in detail. The tests are captured expectations that serve as acceptance / completion criteria and are used to demonstrate a completed implementation.

User stories must address both functional and non-functional characteristics.

User story description examples

Here are some example user stories for a website belonging to an events organising company:

- A user can search for events by location.
- A user can search for events by type.
- A user can book tickets for a single event.
- A user can print a receipt for ticket purchases.

User stories describe functionality that customers will value. It is important then to make sure that user stories are written in the business language so customers will readily understand them. Actually, most user stories are written by the customers anyway. The following examples are not good user stories:

- The program will implement distributed processing to optimise resource use.
- The software shall be easy to maintain.
- The software shall use replaceable components whenever possible.

These may or may not be good goals for development to achieve but they are not good user stories because the customers are unlikely to care whether or not the system implements distributed processing, is easy to maintain or uses replaceable components. These stories do not mention any benefit to the customer.

Story points

The effort required to complete each user story is estimated by the team. This is typically expressed as a number of ‘story points’. The greater the number of story points the greater the estimated effort required to complete the story. The greater the estimated effort, the larger the story is perceived to be. User stories should be small so they can be coded and tested within anything from half a day up to 2 weeks by one or two programmers.

Epics

A large user story is called an epic. Epics are typically broken down into a number of smaller user stories. For example, the user story ‘A user can search for events’ may be considered an epic because there are so many unanswered questions, such as ‘What values can users search on?’, ‘What information is displayed for matching events?’ and ‘Does the user have to be registered first?’.

Acceptance criteria

An essential part of every user story is the acceptance tests. These are the acceptance or completion criteria that reflect the customer expectations. They provide developers and testers with an extended vision of the feature and are used to demonstrate that the user story has been implemented and performs according to the customer expectations. Each story includes acceptance criteria for the functionality as well as any important non-functional characteristics. An Agile team considers a task finished when its associated set of acceptance criteria have been satisfied, that is, all the acceptance tests that have been derived from the acceptance criteria have passed.

The acceptance criteria are defined in collaboration between business representatives, developers, and testers. They help the developers and testers gain a better understanding of the customers' expectations.

The acceptance criteria for a given user story should be written before, but as close to the start of the implementation of the user story as possible. Doing so helps to avoid creating acceptance criteria for a user story that later is changed or reprioritised or removed altogether.

The purposes of acceptance criteria are summarised below:

- help the product owner answer what the business needs in order for this feature to provide value (typically these are the minimum functional requirements)
- define the boundaries for a user story / feature
- help the team gain a shared understanding of the story / feature
- help developers and testers to derive tests
- help developers know when to stop adding more functionality to a story

Example acceptance criteria

User story

> As a conference attendee, I want to be able to register online, so that I can register quickly and cut down on paperwork.

Acceptance criteria

- A user cannot submit a registration form without completing all the mandatory fields.
- Information from the registration form is stored in the registrations database.
- Protection against spam is working.
- Payment can be made via a credit card.
- Payment can be made via PayPal.
- An acknowledgment email is sent to the user after submitting the registration form.

Contribution of the tester

Typically, the tester's unique perspective will improve the user story by identifying missing details or non-functional requirements. A tester will also contribute by asking business representatives questions about the user story, proposing ways to test the user story, and confirming the acceptance criteria.

The INVEST technique

The collaborative authorship of the user story can use techniques such as brainstorming and mind mapping. Testers may use the INVEST technique, which is an acronym for six characteristics of a good user story.

- **I**ndependent

  Stories are easiest to work with if they are independent. Dependent stories can be harder to estimate and may lead to prioritisation and planning issues. For example, if a high priority story is dependent on a lower priority story, which is implemented first?

- **N**egotiable

  A good story is negotiable. That is, there are not a lot of details given thereby necessitating further discussion as the story is implemented. Having a lot of detail written down (as notes of conversations) can give the impression that everything is known about the story and this may supress further discussion that would otherwise highlight additional important information.

  When details are known they should be turned into tests. This leaves the notes to serve as open questions that can be answered during further conversations.

- **V**aluable

Every story needs to be valuable to the customer, meaning an end-user or a purchaser of the software. Stories that are only valuable to developers probably should be reworded so that they express the benefit to the customer. For example, 'The system shall implement load balancing' could be reworded along the lines of 'The system will make optimal use of the system resources for maximum efficiency'.

- **E**stimatable

  A good story can be estimated. Mike Cohn [Cohn04] highlights three reasons for stories being difficult/impossible to estimate: developers lack domain knowledge, developers lack technical knowledge, the story is too big. Where developers lack domain knowledge they need to discuss the story with the customers to gain a better understanding of it. Where they lack technical knowledge they need to spend a little time (by way of a separate story) doing research or experimenting to gain sufficient knowledge to allow a reasonable estimate to be made. Where the story is too big, it should be broken down into a number of smaller stories.

- **S**mall

  Good stories tend to be small, but not too small. Stories can be too small if, for example, writing it down and estimating will take longer than the actual implementation, such as with a minor bug-fix. In this case a number of very small stories can be combined into one story of reasonable size.

  Stories can be too big (and these are called epics) because they are either too complex (a complex story) or they are made up of a number of stories (a compound story). Compound stories can typically be broken down into a number of smaller stories as the details of the compound story are discussed. Complex stories on the other hand are more difficult to break down because of uncertainty – a lack of domain or technical knowledge. In these situations complex stories can be divided into two stories, the first of which undertakes research or experimentation to gain sufficient knowledge to allow the complex user story to proceed.

- **T**estable

  A good story is testable. If a story cannot be tested then the developers can never know when they have finished implementing a story. Stories that are untestable tend to be non-functional stories. For example, 'The user will be able to complete the booking quickly'. This does not define what is meant by 'quickly' so the story must be rewritten to make it testable.

  Good stories will also be mostly testable using automated tests. It is unlikely that all tests will be automatable but for reasons of efficiency and supporting change, most tests should be automated.

See [Web-9] for more details on INVEST.

The 3C concept

User story descriptions are often hand-written on small cards. Because of this Ron Jeffries has named the three parts of a user story as card, conversation and confirmation [Jeffries00]. This is known as the 3C concept.

Card, Conversation, and Confirmation

- Card

  The card is the physical media on which a user story is written. It identifies the requirement, its criticality, expected development and test duration, and the acceptance criteria for that story. The description has to be accurate, as it will be used in the product backlog.

- Conversation

The conversation explains how the software will be used. The conversation can be documented or verbal. Testers, having a different point of view than developers and business representatives, bring valuable input to the exchange of thoughts, opinions, and experiences. Conversation begins during the release-planning phase and continues as the story is implemented.

- Confirmation

  The acceptance criteria, discussed in the conversation, are used to confirm that the story is done. These acceptance criteria may span multiple user stories. Both positive and negative tests should be used to cover the criteria. During confirmation, various participants play the role of a tester. These can include developers as well as specialists focused on performance, security, interoperability, and other quality characteristics. To confirm a story as done, the defined acceptance criteria should be tested and shown to be satisfied.

See [Web-10] and [Web-11] for more information on 3C.

Agile teams vary in terms of how they document user stories. Regardless of the approach taken to document user stories, documentation should be concise, sufficient, and necessary. Consistency of style is also helpful if not important. To help achieve consistency one or more templates may be used, as described below.

User story templates

Templates are often used within Agile development to guide the creation of user stories. (Templates are also commonly used when specifying acceptance criteria.) They are not essential but they are frequently useful. Note that templates are not covered in the syllabus. The information in this subsection is therefore not examinable.

User stories typically follow a simple template such as:

As a <type of user>, I want <some goal> so that <some reason>

e.g. “As a customer, I want to find local events so I can view the details.”

Here the type of user is a ‘customer’ as opposed to a ‘system administrator’ or some other role. It could be made more specific e.g. ‘returning customer’ or ‘new customer’. The goal is ‘to find local events’ implying a need to specify the local area. The reason is to ‘view the details’ of the local events.

Different templates can be used since any one template is unlikely to suit everyone. It is a good idea to experiment with different templates to see which works best. Here are a few more templates with example stories:

As a <role>, I want <function> so that <business value>

e.g. “As a shopper on the website, I want to remove items from my basket so I can change what I buy while shopping.”

As a <role>, I do <action>, so that <result>

e.g. “As a customer I do search for users so that I can view their details.”

<action> <result> <object>

e.g. “Calculate the monthly repayments on the loan.”

“Validate a claim against an insurance policy.”

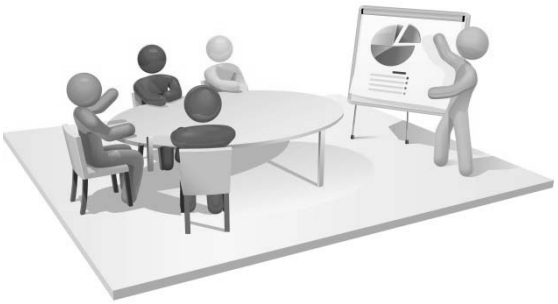
“Turn on the ‘fuel low’ warning light.”

“Assess the risk of an investment.”

1.2.3 Retrospectives

Learning Objective

FA-1.2.3 K2 Understand how retrospectives can be used as a mechanism for process improvement in Agile projects.



In Agile development, a retrospective is a meeting of the whole team held at the end of each iteration to discuss what was successful in the iteration and what could be improved for the next iteration. The meeting will also come up with ideas for how to improve and how to retain the successes.

Retrospectives typically cover topics such as processes, people, organisation, relationships, and tools. Any topic relevant to the iteration can be raised by anyone in the team. However, this must be done in an objective and constructive way. Retrospectives must occur within a professional environment characterised by mutual trust. It is important that everyone feels safe (i.e. will not be criticised) and that every team member is respected. The purpose is not to find fault with people or blame people for things that went wrong, rather it is about identifying where the team as a whole can do better and then implementing a few improvement actions. Doing this at least once every iteration ensures the team improves iteration by iteration.

The attributes of a successful retrospective are the same as those for other reviews, as is discussed in Section 3.2 of the Foundation Level syllabus [ISTQB_FL_SYL].

Being selective

While a retrospective meeting may identify many things that could be improved, in general, teams should implement only a few (possibly just one or two) improvements per iteration. In this way they are more likely to be implemented successfully and the whole improvement effort will not detract from delivering valuable product to the customers. Small but frequent improvements are a good way of achieving continuous improvement at a sustained pace.

There are many ways to select which of the improvement ideas to implement. However it is done it should be done by the whole team in an open and transparent way. Asking everyone to vote for their favourite often does not work if there are many improvement ideas to choose from (this approach may fail to identify an overall winner). Giving each team member ten points to vote with is usually more successful. Team members are allowed to give all ten points to one idea if they believe it stands out from the crowd, or spread the points across a number of ideas.

Retrospectives are critical to self-organisation and continual improvement of both development and testing activities.

Test improvement

Testers should play an important role in the retrospectives. Testers are part of the team and bring their unique perspective (as described in Section 1.5 of the Foundation Level syllabus [ISTQB_FL_SYL]).

Testing occurs in each iteration and vitally contributes to success. Therefore, all team members, testers and non-testers alike, can identify successes and improvement opportunities across all activities including, importantly, testing.

Many test-related improvement ideas are likely to be generated. These may focus on things such as test effectiveness, test productivity, test case quality, and team satisfaction. They may also address the testability of applications, of user stories, of features, and of system interfaces.

As mentioned in the Foundation Level syllabus [ISTQB_FL_SYL] root cause analysis of defects can drive testing and development improvements.

Timing and organisation

The timing and organisation of the retrospective depends on the particular Agile method followed. Business representatives and the team attend each retrospective as participants while the facilitator organises and runs the meeting. In some cases, the teams may invite other participants to the meeting.

There are many approaches to retrospective meetings. One approach entails the team members identifying new activities that should be started, current activities that should be stopped and current activities that should be continued. Each suggestion may be written on a sticky note and stuck to the wall where the whole team can review them. [Crispin08]

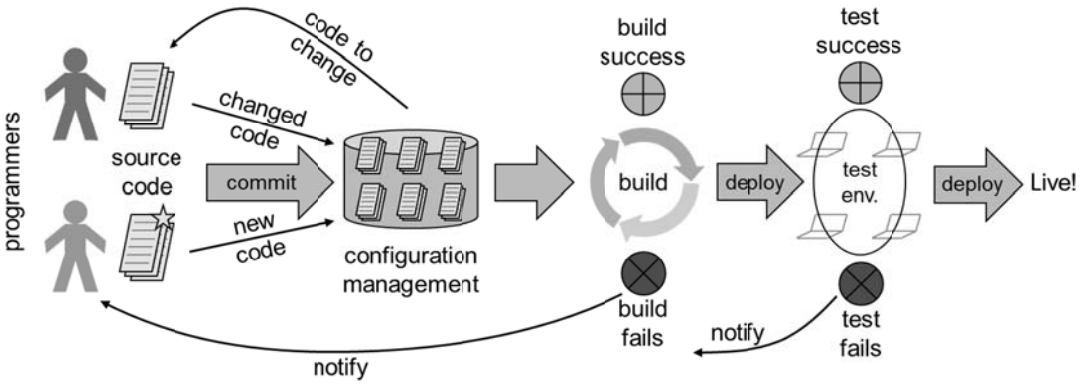
1.2.4 Continuous Integration

Learning Objective

FA-1.2.4 K2 Understand the use and purpose of continuous integration.

Delivery of a product increment requires reliable, working, integrated software at least at the end of every sprint. However, there is a big difference between the situation when a programmer has finished writing code for a new feature and having that feature incorporated into a working version of the product. For example, the following sequence of processes is typical:

1. **Configuration Management:** to start with any new and changed source code files have to be incorporated into the master version of the source code for the whole product. This will involve tasks such as registering new source files and assigning new version numbers to changed versions of existing files.
2. **Compilation:** the source code files are then compiled.
3. **Build:** the files output by the compiler are built into a new executable file.
4. **Deploy:** the new executable file and any associated artefacts are then installed into a test environment.
5. **Testing:** the whole system undergoes a level of testing to confirm that existing functionality remains unaffected by the new changes and that the new changes work as expected.



Continuous integration addresses this challenge by merging all changes made to the software and integrating all changed components regularly, at least once a day. Configuration management, compilation, software build, deployment, and testing are wrapped into a single, automated, repeatable process.

The more frequently the integration of new and changed code files is undertaken the quicker and easier it is to isolate and fix problems caused by incorrect or incomplete code changes. Finding defects as early as possible is the key benefit and this can be improved by performing checks and running tests even before the deployment into the test environment. For example, static analysis of the source code before compilation and unit testing before deployment.

Automating development activities

Following the developers' coding, debugging, and check-in of code into a shared source code repository, a continuous integration process consists of the following automated activities:

- Static code analysis:

  Execute static code analysis and report results.

- Compile:

  Compile and link the code, generate the executable files.

- Unit test:

  Execute the unit tests, measure code coverage and report test results.

- Deploy:

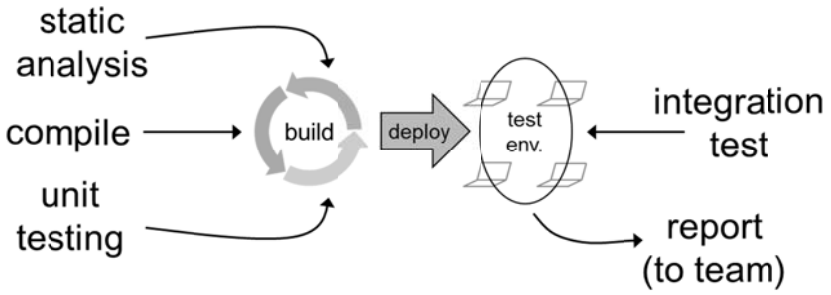
  Install the build into a test environment.

- Integration test:

  Execute the integration tests and report results.

- Report (dashboard):

  Post the status of all these activities to a publicly visible location or e-mail status to the team.



An automated build and test process should take place at least on a daily basis (but could be as frequently as hourly) to detect integration defects early and quickly. When defects are detected in any of the stages outlined above the 'build' is broken, so the process stops until the issues are resolved.

The time it takes to go through this process is critical. For some teams it can be as much as two hours or more. Ideally it should be completed within 10 minutes [Crispin08]. If developers have to wait 2 hours for feedback on changes that they have made and discover that a change did break the build, it will be another two hours before they know if they were able to fix it. Of course, 10 minutes is not a lot of time in which to create and test a new build. The automated process has to be as optimised as possible and some compromises with the testing may be necessary (e.g. run highest priority tests first and continue with additional testing after the initial feedback, or run the less important tests only on the nightly builds).

Some of the early build process steps can be undertaken before the build process itself. For example, code can have static analysis performed on it and it can be compiled immediately a

programmer commits the code to configuration management. Once this is done the automated component tests can be run. In this way there may be little remaining to be done of the pre-deployment stages.

Automated testing

Continuous integration incorporates some automated testing (specifically this must at least cover integration risks). However, continuous integration also allows Agile testers to run additional automated tests regularly. This may be as part of the continuous integration process itself or as a separate activity once the executable files are deployed into the test environment. The aim will always be to send quick feedback to the team on the quality of the code. These test results are visible to all team members, especially when automated reports are integrated into the continuous integration process.

Automated regression testing can be continuous throughout the iteration. Good automated regression tests cover as much functionality as possible, including user stories delivered in the earlier iterations. Good coverage in the automated regression tests helps support building (and testing) large integrated systems. When the regression testing is automated, the Agile testers are freed to concentrate their manual testing on new features, implemented changes, and confirmation testing of defect fixes.

Build tools

In addition to automated tests, organisations using continuous integration typically use build tools to implement continuous quality control.

In addition to running unit and integration tests, such tools can run additional static and dynamic tests, measure and profile performance, extract and format documentation from the source code and facilitate manual quality assurance processes.

This continuous application of quality control aims to improve the quality of the product as well as reduce the time taken to deliver it by replacing the traditional practice of applying quality control after completing all development.

Build tools can be linked to automatic deployment tools, which can fetch the appropriate build from the continuous integration or build server and deploy it into one or more development, test, staging, or even production environments. This reduces the errors and delays associated with relying on specialised staff or programmers to install releases in these environments.

Benefits

Continuous integration can bring the following benefits:

- Allows **earlier detection** and easier root cause analysis of integration problems and conflicting changes.
- Gives the development team **regular feedback** on whether the code is working.
- Keeps the version of the software being tested **within a day** of the version being developed.
- **Reduces regression risk** associated with developer code refactoring due to rapid re-testing of the code base after each small set of changes.
- **Provides confidence** that each day's development work is based on a solid foundation.
- Makes **progress visible**, encouraging developers and testers.
- **Eliminates big-bang integration schedule risks**.
- Provides **constant availability** of executable software throughout the sprint for testing, demonstration, or education purposes.
- **Reduces repetitive manual testing** activities.
- Provides **quick feedback** on decisions made to improve quality and tests.

Risks and challenges

However, continuous integration is not without its risks and challenges:

- Continuous integration **tools** have to be introduced and maintained.
- The continuous integration **process** must be defined and established.
- Test automation requires **additional resources** and can be complex to establish.
- Thorough **test coverage** is essential to achieve automated testing advantages.
- Teams sometimes **over-rely on component tests** and perform too little system and acceptance testing.

Continuous integration requires the use of tools, including tools for testing, tools for automating the build process, and tools for version control.

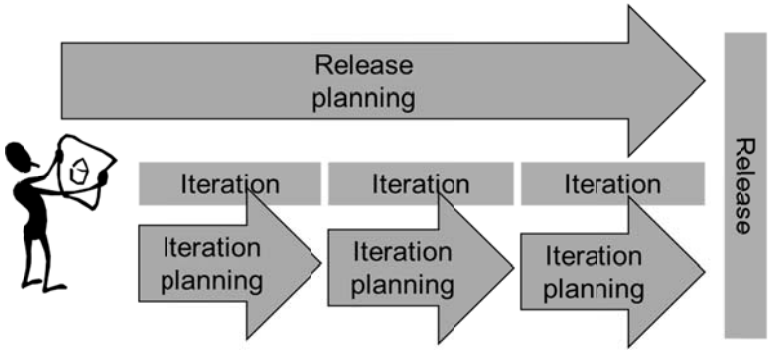
1.2.5 Release and Iteration Planning

Learning Objective

FA-1.2.5 K1 Know the differences between iteration and release planning, and how a tester adds value in each of these activities.

As mentioned in the Foundation Level syllabus [ISTQB_FL_SYL], planning is an on-going activity, and this is the case in Agile lifecycles as well. For Agile lifecycles, two kinds of planning occur, release planning and iteration planning.

Release planning occurs at the very start of the project and looks ahead to the release of the product, often in a few months' time. **Iteration planning** occurs at the very start of each iteration and looks ahead to the end of that iteration, typically in one or two weeks' time.



Release planning

Release planning is concerned with the product backlog. Release planning meetings involve the whole Agile team and can include others by invitation from the team (e.g. specialists in areas relevant to the feature set that this release is to produce). They work together to gain a common understanding of the customer's vision of what is to be delivered, its purpose and value to customers. They talk about the features and stories to gain a high-level view of what may be included in the release and how many iterations may be required.

Business representatives establish and prioritise the user stories for the release in collaboration with the team (see Section 1.2.2 "Collaborative User Story Creation"). Based on these user stories, project and quality risks are identified and high-level effort estimation is performed, dependencies between stories are considered along with other factors that influence the order in which they may be implemented. Large user stories (epics) may be split into a collection of smaller stories. In these ways, release planning defines and re-defines the product backlog.

Release planning also provides the basis for a test approach and test plan spanning all iterations. Release plans are high-level.

During release planning general questions will be raised and answered. Questions about the project such as if other teams will be involved, how they will be coordinated, what metrics are necessary to track progress, etc.

Testers' involvement in release planning

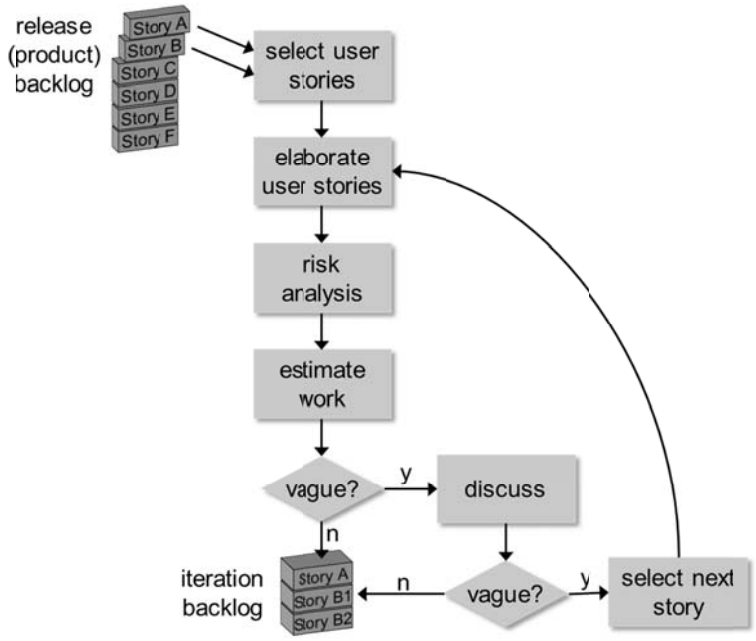
Release planning gives the developers, testers and customers the opportunity to consider the impact of the planned features on the larger system, to clarify assumptions and identify what resources may be necessary (skills, test environments, tools, etc.). Testers in particular are likely to add value in the activities listed below.

- Defining testable user stories, including acceptance criteria.
- Participating in project and quality risk analysis.
- Estimating testing effort associated with the user stories.
- Defining the necessary test levels and test types.
- Planning the testing for the release.

Iteration planning

After release planning is done, iteration planning for the first iteration starts.

Iteration planning is concerned with the iteration backlog. Iteration planning meetings involve the whole Agile team and can include others by invitation from the team.



As shown in the diagram above, the team:

- selects user stories from the prioritised release backlog,
- elaborates the user stories,
- performs a risk analysis for the user stories, and
- estimates the work needed for each user story.

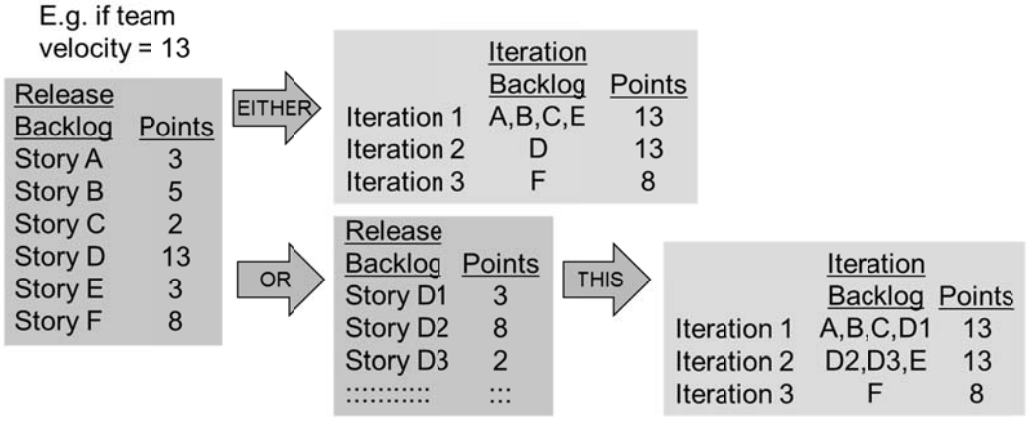
Estimates for the work needed to implement a story are often given in 'story points'. The estimate is a relative indication of the size and complexity of the story (relative to other stories). So a user story estimated to be 10 story points is expected to take twice as long as a story estimated to be 5 story points. Large estimates imply that the story is too big and should be broken down into a collection of smaller stories. The story points are equated to a meaningful measure such as 'effort days'. See section 3.2.2 "Estimating Testing Effort Based on Content and Risk".

If a user story is too vague and attempts to clarify it have failed, the team can refuse to accept it and instead use the next user story based on priority. The business representatives must answer the team's questions about each story so the team can understand what they should implement and how to test each story.

After the contents of the iteration are finalised, the user stories are broken into tasks that are then assigned to the appropriate team members.

Selecting stories

The number of stories selected is based on an established (or in the case of the first iteration, an estimated) team velocity and the estimated size of the selected user stories. The team velocity is the number of story points (or other units) that the team believes they can complete in one iteration. This will be based on the actual number of story points implemented (the team's 'velocity') in the previous iterations (or an estimate in the case of the first iteration).



In the example above we have a team velocity of 13 so each iteration can include stories from the release backlog with a total of no more than 13 points. Taking the first 3 stories gives us a total of 10 points but we can't include story D as this is itself 13 points. Either we can take the next story (story E) which gives us a total of 13 or we can divide story D into a set of smaller stories and include the first of these into the first iteration.

Testers' involvement in iteration planning

Testers are involved in iteration planning and especially add value in the following activities listed below.

- Determining the testability of the user stories.
- Participating in the detailed risk analysis of user stories.
- Creating acceptance tests for the user stories.
- Breaking down user stories into tasks (particularly testing tasks).
- Estimating testing effort for all testing tasks.
- Identifying functional and non-functional aspects of the system to be tested.
- Supporting and participating in test automation at multiple levels of testing.

Changes to release plans

Release plans may change as the project proceeds, including changes to individual user stories in the product backlog. These changes may be triggered by internal or external factors.

Internal factors include:

- delivery capabilities
- velocity
- technical issues.

External factors include:

- the discovery of new markets and opportunities
- new competitors
- business threats that may change release objectives and / or target dates.

In addition, iteration plans may change during an iteration. For example, a particular user story that was considered relatively simple during estimation might prove more complex than expected.

These changes can prove challenging for testers. Testers must understand the big picture of the release for test planning purposes, that is, understand the value to the customer and be able to distinguish between what is essential and is not.

Testers must also have an adequate test basis and test oracle (though not necessarily in the form of a written specification) in each iteration for test development purposes. The business representatives should be on hand to help with this but the testers cannot rely on them for every test design decision.

The required information must be available to the tester early, and yet change must be embraced according to Agile principles. This dilemma requires careful decisions about test strategies and test documentation.

For more on Agile testing challenges, see [Black09], Chapter 12.

Test planning

Release and iteration planning should address test planning as well as planning for development activities. Particular test-related issues to address include:

- The **scope of testing**, the extent of testing for those areas in scope, the test goals, and the reasons for these decisions.
- The **team members** who will carry out the test activities.
- The **test environment and test data needed**, when they are needed, and whether any additions or changes to the test environment and / or data will occur prior to or during the project.
- The **timing, sequencing, dependencies, and prerequisites** for the functional and non-functional test activities (e.g., how frequently to run regression tests, which features depend on other features or test data, etc.), including how the test activities relate to and depend on development activities.
- The **project and quality risks** to be addressed (see Section 3.2.1).

In addition, the larger team estimation effort should include consideration of the time and effort needed to complete the required testing activities.

1.3 Summary

The Agile Manifesto puts more value on individuals, interactions, working software, customer collaboration and responding to change. These values are supported by 12 principles.

A key feature of Agile development is the whole-team approach. Agile teams are made up of programmers, testers and business representatives / customers. They work together throughout the project and ideally are co-located, enhancing collaboration and communication. Quality is the responsibility of everyone on the team.

Another key feature of Agile development is the short iterations that each deliver a potentially shippable product. This provides early and frequent feedback on the quality of the code, avoiding requirement misunderstandings and clarifying customer feature requests.

There is not a standard approach to Agile development but there are a number of methods that each implement the Agile Manifesto in different ways. We outlined three of the more common methods: Extreme Programming, Scrum and Kanban.

Essential practices used with Agile development include collaborative user story creation, retrospectives, continuous integration and release and iteration planning.

Collaborative user story creation combats the traditional communication misunderstandings that result in products being delivered that do not match the expectations or needs of the customers. The requirements are expressed in terms of short descriptions of a specific feature along with acceptance criteria and on-going conversations between the customers, developers and testers. This is also known as the 3C concept: Card, Conversation and Confirmation. Characteristics of a good user story are described by the INVEST acronym: Independent, Negotiable, Valuable, Estimatable, Small and Testable.

Retrospectives are meetings held at the end of each iteration involving the whole team identifying what was successful and what can be improved. One particular technique known as the Start, Stop, Continue technique has the team identify new activities that should be started, and existing activities that should be stopped or continued. Although many ideas may be raised, only a few should be implemented in the next iteration: small but frequent improvements.

Continuous integration is the frequent (not less than daily) build cycles that deliver the latest software changes for more detailed testing and possible delivery to the customers. The build process must be automated and should include early checking and testing such as static analysis, component testing and integration testing. Ideally the build process should complete within 10 minutes. The benefits of continuous integration include earlier detection and easier fixing of integration defects, reduced regression risk, more visible progress and quicker feedback.

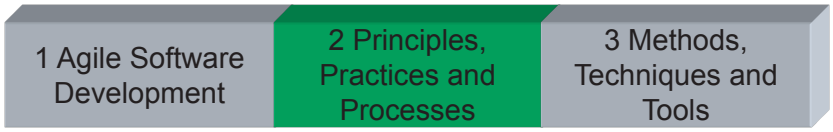
Release planning involves the whole team and is concerned with the product backlog – identifying user stories that are to be included in the release. The team discuss the user stories to gain a high-level understanding of the customers' vision and estimates of required effort for each story at a high level.

Iteration planning also involves the whole team and is concerned with the iteration backlog – selecting user stories for inclusion in the next iteration. More detailed estimation is undertaken. Both types of planning include test planning, identifying the scope of testing, assigning test activities to team members, establishing the test environment, timing and dependencies of user story testing.

ISTQB Foundation Extension Agile Tester

Module 2:
Fundamental Agile Testing Principles,
Practices and Processes

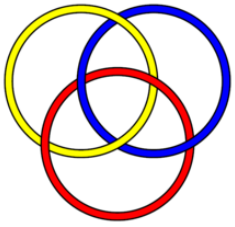
Fundamental Agile Testing Principles, Practices and Processes



Contents

Testing and development activities

- testing and development activities are interconnected
  - structures defined by models that are:
    - ► sequential, iterative and/or incremental
  - Agile is 'iterative incremental'



- Agile models differ:
  - how activities are integrated
  - project work products
  - names
  - test level entry/exit criteria
  - use of tools
  - use of independence in testing
- ability to adapt is key
  - lifecycle implementations vary greatly
  - deviations from the ideals
    - ► may be intelligent customisation and adaption

4

Contents

5

Comparing testing and development activities 1

Agile

- short iterations
  - usually 1 to 4 weeks
  - implements a few user stories
  - early and frequent feedback
- working software
  - deliverable of each iteration
  - decision based on quality

Traditional

- longer iterations
  - few months if not longer
  - major features
  - feedback later and much less frequent
- delivered software
  - delivery deadline sometimes more important than quality

6

Comparing testing and development activities 2

Agile

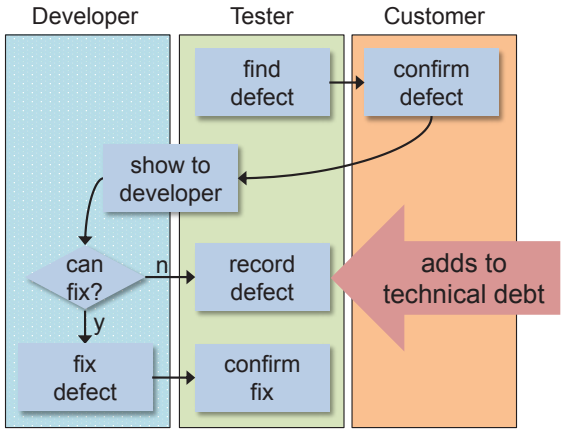
- testing roles
  - developers, testers and business stakeholders
    - ► all involved in testing throughout each iteration
    - ► most likely from unit tests to acceptance tests inclusive
- technical debt
  - strategies to resolve early or avoid altogether

Traditional

- testing roles
  - developers: unit test (if time permits)
  - testers: other test levels (often subject to 'squeeze')
  - business stakeholders often not involved in testing
- technical debt
  - accumulates

7

Managing defects in Agile



8

Comparing testing and development activities 3

Agile

- risk-based testing
  - risk analysis during release and iteration planning
  - influences sequence of development and priority and depth of testing
- pairing
  - continuous review
  - shared ownership of quality

Traditional

- risk-based testing
  - emphasis on risk analysis at start of project
  - on-going risk analysis with view of whole project
- pairing
  - not traditional

9

Comparing testing and development activities 4

Agile

- sharing knowledge
  - testers serve as testing and quality coaches
- test automation
  - extensive at all test levels
  - higher % non-systematic techniques
- change
  - is embraced
  - lightweight documentation

Traditional

- sharing knowledge
  - often departments work in silos
- test automation
  - can be extensive at all levels
  - often not until product is stable
- change
  - not welcomed
  - costly

10

Contents

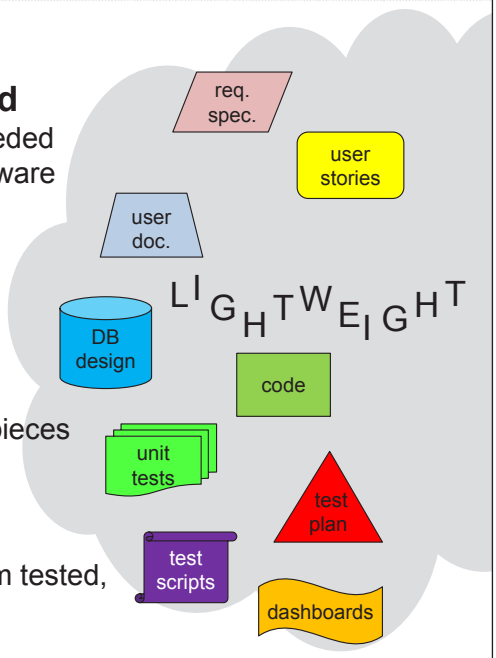
11

Project work products

business-oriented
describe what is needed
and how to use software

development
describe how built,
that implement, and
that evaluate code pieces

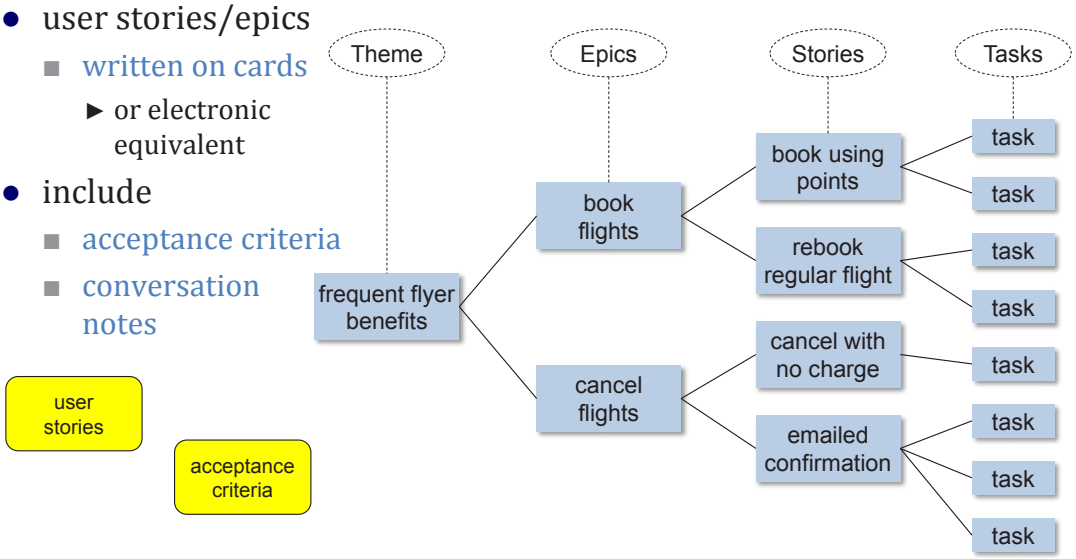
testing
describe how system tested,
that test, and
that present results



12

Business-oriented work products in Agile

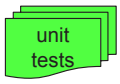
- user stories/epics
  - written on cards
    - ► or electronic equivalent
- include
  - acceptance criteria
  - conversation notes



13

Development work products in Agile

- code
  - low technical debt
    - ► no known defects
    - ► easy to maintain
    - ► testable
- unit tests
  - manual or automated
  - form of executable low-level design specification (TDD)



test-driven development (TDD): automated unit tests created and run before code is written to pass tests one test at a time

14

Tester work products in Agile

- tests
  - manual and automated
- test plans/strategies
- defect reports
- test result logs
- test reports
  - including metrics

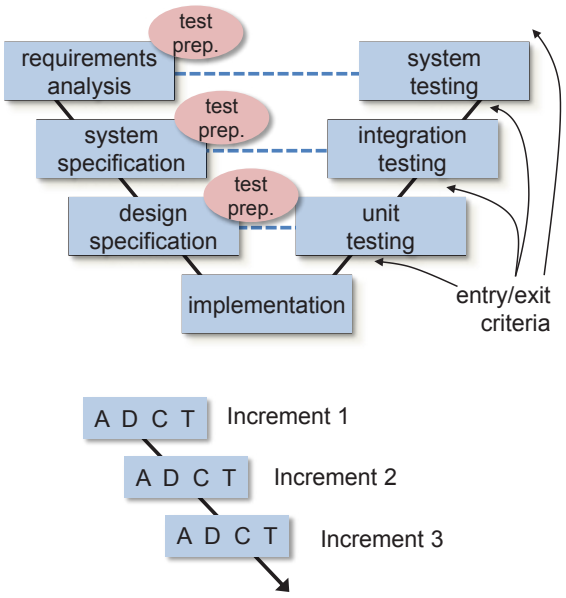
15

Contents

16

Test levels in sequential and incremental models

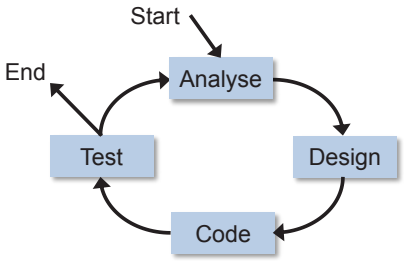
- sequential models
  - test levels are sequential
  - test preparation overlaps development activities
  - entry/exit criteria for each level
- incremental models
  - test levels (within increment) overlap
  - test preparation overlaps development activities
  - increments may overlap

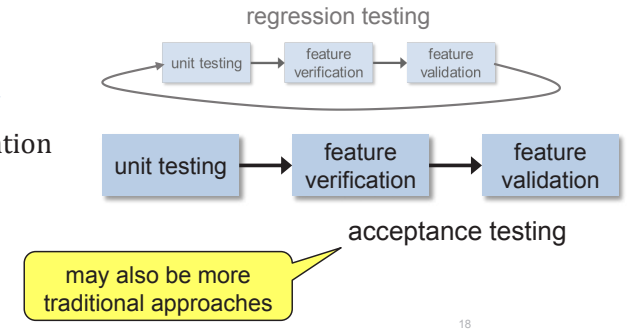


17

Test levels in iterative and Agile models

- iterative models
  - test levels (within iteration) overlap
  - test preparation overlaps development activities
- Agile models
  - automated unit testing by developers
  - feature acceptance testing
    - ► automated feature verification
    - ► manual feature validation



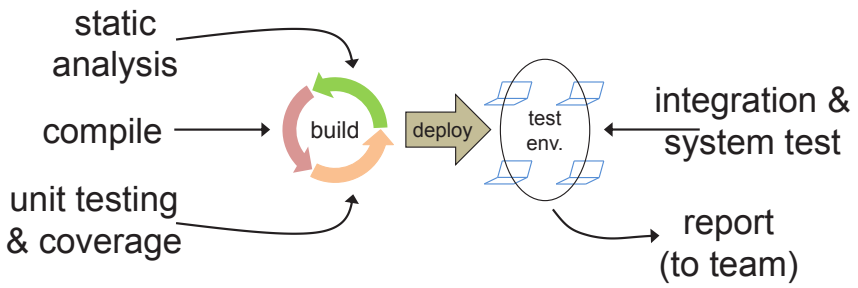


18

Contents

19

Testing and configuration management



- tool support crucial for Agile
  - static analysis
  - unit testing
  - code coverage
- test failures
  - fix defects before next code check-in
- mitigate regression risk
  - include functional integration and system testing
    - ► may be run less frequently

20

Contents

21

Independence in testing

Independent testers typically find some different defects from those found by testers with more familiarity with the software.

It is therefore good practice to include some independent testers when possible to increase the overall effectiveness of the testing.

Remember: it is who designs the tests, not so much who runs them that counts here.

22

Organisational options for independent testing

- developers automate
  - verification less error prone
- testers embedded
  - full-time testers on Agile team
  - risk of losing independence
- on-demand testers
  - maintain separate test teams
  - assign testers on-demand
    - ► during final days of iteration
  - possible problems
    - ► time pressure
    - ► lack of understanding
    - ► relationship issues
- separate test team
  - testers assigned to Agile teams at start of project
    - ► maintain independence
    - ► gain good understanding
    - ► build relationships
  - specialised testers
    - ► iteration independent
      - – developing automation tools
      - – non-functional testing
      - – test environments
      - – other test levels

23

Contents

Status of testing in Agile projects

- change happens rapidly
  - test status
  - test progress
  - product quality
    - ► all evolve
- latest situation must be communicated to the team
  - which user stories are done?
  - which tasks remain?
  - which tests have failed?
- help team make decisions
  - to stay on track
  - successful completion

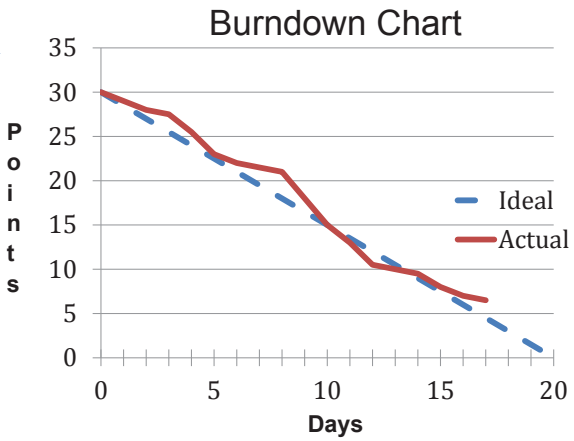
Contents

Communicating test status

- goal is working software
  - at end of fixed length iteration
- monitoring progress essential
  - must know during iteration
    - ► what's done
    - ► what's outstanding
    - ► product quality
- methods include
  - test automation results
  - burndown charts
  - Agile task board
- test results
  - important way to measure progress
    - ► charts showing number of tests created, run, passed

Burndown charts

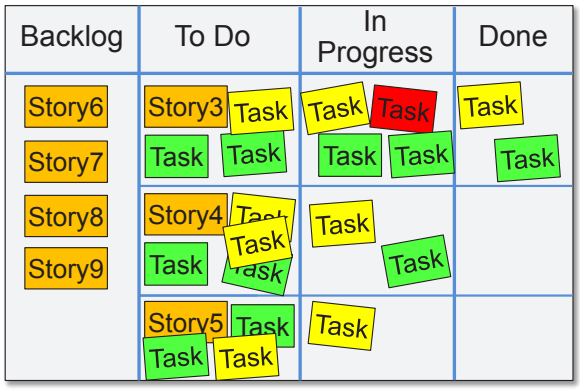
- work remaining vs. time allocated to complete
  - based on estimates made during release and iteration meetings
- used to track progress



28

Agile task board

- detailed representation of whole team's current status
- include
  - story cards, development tasks, testing tasks, defects and other tasks
    - ► cards colour co-coded
      - – e.g. story cards: brown, test tasks: yellow, etc.
- columns represent stages
  - to do, in progress, done
- rows represent stories
  - those being implemented



29

Stand-up meetings, surveys and metrics

- stand-up meetings
  - includes the whole team
    - ► each person reports status of their work
      - – what has been completed since last meeting?
      - – what is planned before next meeting?
      - – what is getting in the way?
    - ► whole team helps resolve issues
- customer satisfaction surveys
  - used to gain feedback on how well the product meets customer expectations
- other metrics
  - as for traditional projects useful metrics include
    - ► test pass/fail rates
    - ► defect found/fixed rates
    - ► defect density
    - ► risk coverage
    - ► code coverage/churn

to improve product quality

30

Contents

31

Managing regression risk

- regression risk
  - unintended impact on existing features
    - ► new features
    - ► changed features
    - ► removing technical debt
  - grows with each iteration

code churn

- code churn
  - number of lines of code
    - ► added, modified, deleted
- automated regression testing
  - during every iteration
  - essential
  - any failure is investigated
    - ► defect fix
    - ► test removed (or archived)
    - ► new acceptance criteria

"No Agile team will succeed by doing only manual testing." Lisa Crispin

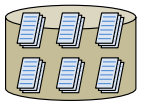
32

Test asset maintenance

- test assets
  - automated (and manual) tests
    - ► test data
    - ► test scripts
    - ► expected results
    - ► etc.
- must be maintained
  - keep up-to-date with each iteration
- also automate more tests

technical debt

- how team designs, writes and stores testware
  - affects maintainability, usability, etc.
- configuration management
  - identifies dependencies
  - version control
  - ease of access



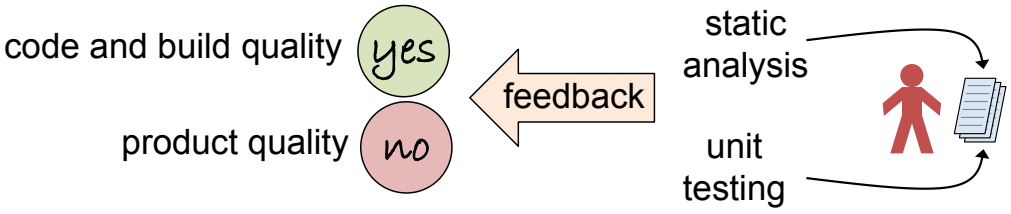
configuration management

- automated tests
  - living document of system functionality

33

Approaches to managing regression risk

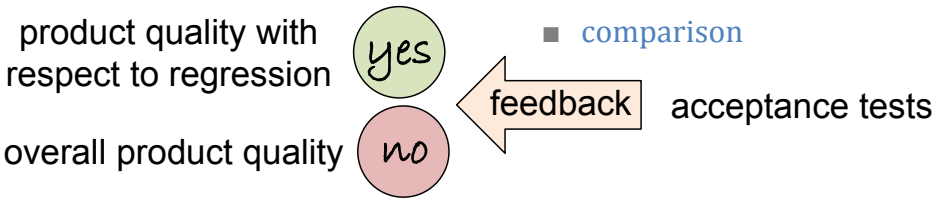
- review test cases
  - identify candidates for regression test suite
    - ► automated and manual
  - identify redundant tests
    - ► to remove
  - aim: reduce regression test effort
- test before check-in
  - developers run static analysis and unit tests before code check-in
  - prevents code changes breaking the build
    - ► may delay whole team



34

More ways to manage regression risk

- build verification tests
  - subset of automated tests
    - ► covers critical system functionality and integration points
  - run immediately on new build
  - prevents further testing on an unstable build
- automated acceptance tests
  - run on complete system build
    - ► at least daily
    - ► generally not with every build
- automate other testing tasks
  - data generation & loading
  - deploy builds into test env.
  - test environment restores
  - comparison



35

Contents

36

Tester skills (any development lifecycle)

- analysis of test basis — analytical skills, attentive to detail
- design of tests — problem solving
- definition of expected results — applying domain knowledge
- test execution and verification — diligent, persistent
- recording and reporting — written and verbal communication skills
- participation in risk management activities — interpersonal skills
- involvement with test automation — test automation skills

37

Competencies for Agile testers

- test automation
- test-driven development (TDD)
- acceptance test-driven development (ATDD)
- white-box (structure-based) testing
- black-box (behaviour-based) testing
- experience-based testing

Also behaviour-driven development (BDD)? (see section 3.1.1)

Including exploratory testing

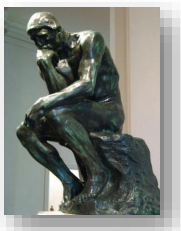
Functional and non-functional testing

38

Skills of particular importance to Agile testers

- be positive and solution-oriented
  - ► with team members and stakeholders

- display critical, quality-oriented, sceptical thinking
  - ► about the product



- accurately evaluate and report
  - ► test results, test progress, and product quality

39

Skills more specific to Agile testers

- actively acquire information from stakeholders
  - ► rather than relying on written specifications
- work effectively to define testable user stories especially acceptance criteria
  - ► with customer representatives and stakeholders
- collaborate within the team
  - ► working in pairs with programmers and other team members
- respond to change quickly
  - ► including changing, adding or improving test cases
- plan and organise their own work

40

Contents

41

Some testing tasks for an Agile tester

- test strategy
  - understand, implement, update
- test coverage
  - measure, report
- test tools
  - ensure proper use
- test environments
  - configure, use, manage
- defects
  - report, help resolve

42

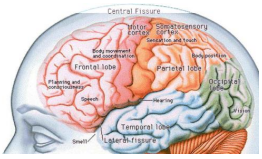
More testing tasks for an Agile tester

- coaching testing
  - other team members
- testing tasks
  - appropriate, scheduled
- clarify requirements
  - collaborating with developers and business stakeholders
- team retrospectives
  - participating, suggesting and implementing improvements

43

Test-related organisational risks in Agile projects

- risk 1
  - testers may lose tester mind-set
    - ► while working so closely with developers
- risk 2
  - testers may become tolerant of, or silent about,
    - ► inefficient, ineffective or low-quality practices
- risk 3
  - testers may not keep pace with the rate of change



Summary

- traditional vs. Agile models
  - ► short iterations
  - ► working software
  - ► collaborative approach
  - ► minimise technical debt
  - ► pairing
  - ► sharing knowledge
  - ► test automation
  - ► embracing change
  - ► lightweight documentation
- test levels
  - overlap, no entry/exit criteria
  - acceptance testing at the end of one or a series of iterations
- configuration management
  - key to build process and continuous integration
- communicating test status
  - ► automated test results
  - ► task boards
  - ► burndown charts
- managing regression risk
  - automated tests, maintain test assets, review tests
- Agile tester skills and roles
  - wider range, some more important

ISTQB Foundation Extension Agile Tester

Session 2:
Fundamental Agile Testing Principles,
Practices and Processes

Terms
build verification test, configuration item, configuration management.

From the ISTQB Glossary

build verification test: A set of automated tests which validates the integrity of each new build and verifies its key/core functionality, stability and testability. It is an industry practice when a high frequency of build releases occurs (e.g., agile projects) and it is run on every new build before the build is released for further testing. See also regression testing, smoke test.

configuration item: An aggregation of hardware, software or both, that is designated for configuration management and treated as a single entity in the configuration management process.

configuration management: A discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements.

test level: A group of test activities that are organized and managed together. A test level is linked to the responsibilities in a project. Examples of test levels are component test, integration test, system test and acceptance test.

2.1 The Differences between Testing in Traditional and Agile Approaches

Learning Objectives

FA-2.1.1	K2	Describe the differences between testing activities in Agile projects and non-Agile projects.
FA-2.1.2	K2	Describe how development and testing activities are integrated in Agile projects.
FA-2.1.3	K2	Describe the role of independent testing in Agile projects.

Testing is an important part of software development and, like development, testing is not a single process or activity but a set of processes and activities. These development and testing processes are mixed together throughout the software lifecycle. Without the development processes, testing means nothing, as the testing processes 'test' what the development processes produce. Testing processes support the development processes; without testing development will not succeed.

The testing processes are interconnected with the development processes. There are many different ways of structuring the development and testing processes but there are some common structures and these are described by software development lifecycle models that fall into one or more of the following three categories.

- Sequential e.g. Waterfall model, V-model and W-model
- Iterative e.g. Rapid Application Development (RAD) and Spiral model
- Incremental e.g. Evolutionary methods

The ISTQB Glossary defines Agile as an iterative incremental approach.

The various Agile models differ in a number of ways, including:

- the way testing and development activities are integrated,
- the project work products,
- the names,
- entry and exit criteria used for various levels of testing,
- the use of tools, and
- how independent testing can be used effectively.

When moving between different lifecycle models, testers must understand the differences between testing in the more traditional lifecycle models and Agile models in order to work effectively and efficiently.

Adapting Agile

Testers should remember that organisations vary considerably in their implementation of lifecycles. Deviation from the ideals of Agile lifecycles (see Section 1.1 "The Fundamentals of Agile Software Development") may represent intelligent customisation and adaptation of the practices. The ability to adapt to the context of a given project, including the software development practices actually followed, is a key success factor for testers.

2.1.1 Testing and Development Activities

Learning Objectives

FA-2.1.1 K2 Describe the differences between testing activities in Agile projects and non-Agile projects.

Short iterations and working software

One of the main differences between traditional lifecycles and Agile lifecycles is the concept of very short iterations with each iteration resulting in working software that can potentially be delivered and therefore provide features of value to business stakeholders. The iterations are usually between 1 and 4 weeks. The aim of every iteration is the production of working software but the decision to deliver is made at the end of the iteration by the business representatives and on the basis of the quality of the software (i.e. does it provide the required value).

At the beginning of the project, there is a release planning period as outlined in Section 1.2.5 “Release and Iteration Planning”. This is followed by a sequence of fixed length iterations. At the beginning of each iteration, there is an iteration planning period to establish which user stories are to be implemented during the iteration. Once iteration scope is established, the selected user stories are developed, integrated with the system, and tested.

These iterations are highly dynamic, with development, integration, and testing activities taking place throughout each iteration, and with considerable parallelism and overlap.

Testing activities (including test execution) occur throughout the iteration, not as a final activity. This avoids the ‘testing squeeze’ that traditional lifecycles often suffer from, usually as a result of the earlier project activities taking longer than planned and the release date remaining fixed.

The team builds and tests the code a little at a time, ensuring that it works correctly before moving on to the next piece of code. Developers never create code that goes into a queue waiting to be tested because the implementation of a user story is not complete (referred to as ‘done’) until it has been tested successfully.

At the end of the sequence of iterations, there can be a set of release activities to ready the software for delivery, though in some cases delivery occurs at the end of each iteration.

Testing roles within Agile

Testers, developers, and business stakeholders all have a role in testing in Agile lifecycles as they do with traditional lifecycles. However, with Agile all the testing roles are on-going throughout the release and are much more collaborative (the Agile team members work together to undertake their various testing roles).

Business stakeholders identify acceptance criteria that are then expanded into acceptance tests in collaboration with the testers. Throughout the project business stakeholders contribute to the development and testing activities by answering questions and confirming the suitability of tests.

As developers implement features from the user stories they create and execute component (unit) tests. Many of these component tests (perhaps all) will be automated.

In collaboration with the business stakeholders, testers expand the acceptance tests and identify any non-functional testing requirements. Some of the tests may be automated while others are run manually. In addition exploratory testing may be undertaken by testers and/or business stakeholders. Business stakeholders might use written test cases, but they also might simply experiment with and use the feature in order to provide fast feedback to the development team.

Resolving technical debt

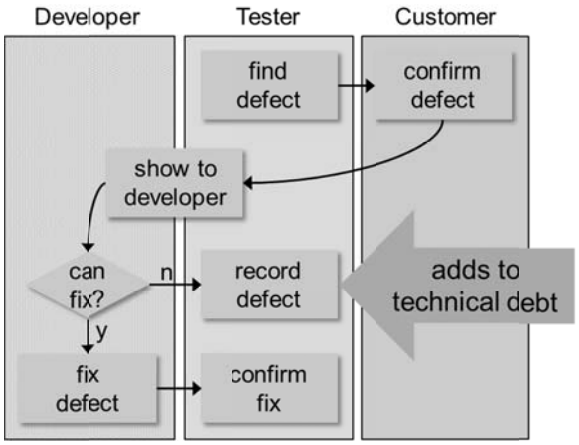
Technical debt builds up when the development team compromise on aspects of the software that typically do not have an immediate impact. For example they may take shortcuts when coding to produce code that works but is not as well-structured or documented. This makes the code harder to work with next time, increasing maintenance effort and reducing the team's efficiency (velocity in Agile terms). Outstanding defects are a common form of technical debt. Not automating tests is another. Programmers become afraid to make changes because it is more difficult to understand and will take more effort to regression test.

Technical debt can be managed in a number of ways. In some cases, hardening or stabilisation iterations can be undertaken. (This is an iteration that has the sole purpose of removing technical debt rather than adding new features.) Another good practice is to address defects remaining from the previous iteration at the beginning of the next iteration, as part of the backlog for that iteration (referred to as "fix bugs first"). However, some complain that this practice results in a situation where the total work to be done in the iteration is unknown and it will be more difficult to estimate when the remaining features can be done.

Perhaps the best approach is to avoid technical debt as much as possible in the first place. Adopting the rule "no feature is considered done until it has been integrated and tested with the system" can be particularly effective. See [Goucher09], Chapter 15.

The diagram on the right shows a possible process for dealing with defects within an Agile team and the respective roles.

In this case the tester finds a defect and confirms that it is a defect with the customer (business representative). The tester then shows the defect to a developer who determines if it can be fixed in the current iteration. If yes, the defect is fixed and the tester runs tests sufficient to confirm the fix. If no, the tester records the defect in sufficient detail so it can be picked up and fixed in a later iteration.



Risk-based testing

When risk-based testing is one of the test strategies, a high-level risk analysis occurs during release planning, with testers often driving that analysis. However, the specific quality risks associated with each iteration are identified and assessed, in iteration planning. This risk analysis can influence the sequence of development as well as the priority and depth of testing for the features. It also influences the estimation of the test effort required for each feature (see Section 3.2).

Pairing

In some Agile practices (e.g., Extreme Programming), pairing is used. Pairing involves team members working together as a pair to complete a single task within a short time-boxed period (often called a paired session and lasting between 30 and 90 minutes). Pairing can involve two testers or two developers. It can also involve a tester working collaboratively with a developer to develop and test a feature.

Pairing has the advantage of a built-in continuous review, as one person types at the keyboard the other person assesses what is being created, encouraging and challenging in equal measure. Having someone else to discuss ideas as they occur makes for quick (if not instant) feedback and early (if not immediate) defect detection.

Pairing can be difficult when the test team is distributed and even more difficult when the team is split across different time zones. Although more difficult in these circumstances, it is not impossible. A good process and some tools can help enable distributed pairing.

For more on issues associated with distributed work, see [ISTQB_ALTM_SYL], Section 2.8.

Sharing knowledge

Testers may also serve as testing and quality coaches within the team, sharing testing knowledge and supporting quality assurance work within the team. This can help to promote a sense of collective ownership of quality of the product.

Test automation

Often extensive test automation is undertaken at all levels of testing in Agile teams. As a consequence testers may spend time creating, executing, monitoring, and maintaining automated tests and results. Because of the heavy use of test automation, a higher percentage of the manual testing on Agile projects tends to be done using experience-based and defect-based techniques such as software attacks, exploratory testing, and error guessing.

(See [ISTQB_ALTA_SYL], Sections 3.3 and 3.4 and [ISTQB_FL_SYL], Section 4.5.)

While developers will focus on creating automated component (unit) tests, testers should focus on creating automated integration, system, and system integration tests. This leads to a tendency for Agile teams to favour testers with a strong technical and test automation background.

Change

One core Agile principle is that change may occur throughout the project. Therefore lightweight work product documentation is favoured in Agile projects. Changes to existing features have testing implications, especially regression testing implications. The use of automated testing is one way of managing the amount of test effort associated with change. However, it's important that the rate of change does not exceed the project team's ability to deal with the risks associated with those changes.

2.1.2 Project Work Products

Project work products of immediate interest to Agile testers typically fall into three categories:

1. Business-oriented work products

   These describe what is needed (e.g. requirements specifications) and how to use it (e.g., user documentation).

2. Development work products

   These describe how the system is built (e.g., database entity-relationship diagrams), that actually implement the system (e.g., code), or that evaluate individual pieces of code (e.g., automated unit tests).

3. Test work products

   These describe how the system is tested (e.g., test strategies and plans), that actually test the system (e.g., manual and automated tests), or that present test results (e.g., test dashboards as discussed in Section 2.2.1).

In a typical Agile project, it is a common practice to avoid producing vast amounts of documentation. Instead, focus is more on having working software, together with automated tests that demonstrate conformance to requirements. This encouragement to reduce documentation applies only to documentation that does not deliver value to the customer.

In a successful Agile project, a balance is struck between increasing efficiency by reducing documentation and providing sufficient documentation to support business, testing, development, and maintenance activities. The team must make a decision during release planning about which work products are required and the level of work product documentation needed.

More formalised projects

In some Agile environments, especially regulated, safety critical, distributed, or highly complex projects and products, further formalisation of these work products may be required. For example, some teams transform user stories and acceptance criteria into more formal requirements specifications. Vertical and horizontal traceability reports may be prepared to satisfy auditors, regulations, or other requirements.

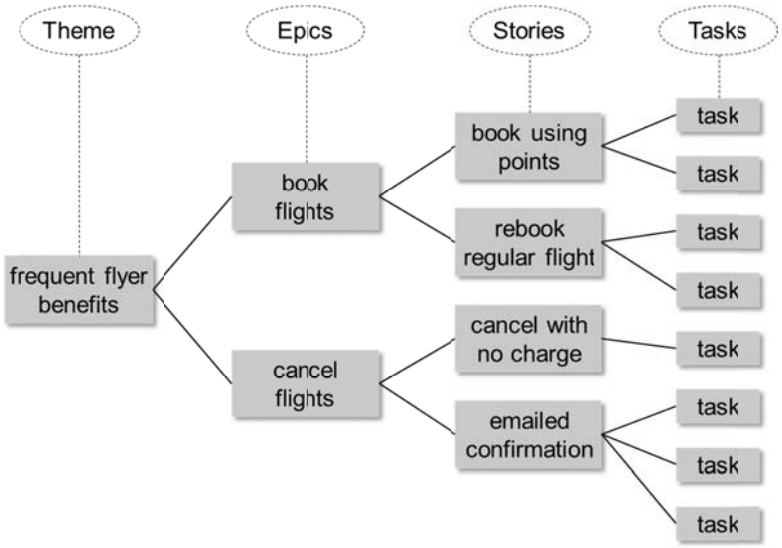
Typical business-oriented work products

Typical business-oriented work products on Agile projects include user stories and acceptance criteria.

User stories are the Agile form of requirements specifications, and should explain how the system should behave with respect to a single, coherent feature or function. A user story should define a feature small enough to be completed in a single iteration.

Larger collections of related features, or a collection of sub-features that make up a single complex feature, may be referred to as "epics". Epics may include user stories for different development teams. For example, one user story can describe what is required at the API-level (middleware) while another story describes what is needed at the UI-level (application). These collections may be developed over a series of sprints. Each epic and its user stories should have associated acceptance criteria.

A collection of epics is known as a "theme". This may also be referred to as the 'big picture'. The diagram below shows the relationship between a theme, epics, user stories and tasks.



Typical developer work products

Perhaps the main developer work product on Agile projects is the program code. However, Agile developers also often create automated unit tests. These automated unit tests may be created after the development of code, though where test-driven development is practiced the automated unit tests will be created incrementally before each portion of the code is written. This provides a way of verifying whether the code works as expected as soon as it is written. The automated tests are run before the code is written to demonstrate that they do fail. The code is then written to pass the tests, one test at a time.

Note that the syllabus states that test-driven development (TDD) is also known as test-first development. Gerard Meszaros [Meszaros07], pp. 813-814 explains how test-first development differs from test-driven development: "Test-first development merely says the tests are written before the production code; it does not imply that the production code is made to work one test at a time."

With test-driven development the unit tests are a form of executable low-level design specification [Beck02].

Typical tester work products

Typical tester work products on Agile projects include automated tests, as well as documents such as test plans, quality risk catalogues, manual tests, defect reports, test reports and test results logs.

The documents are captured in as lightweight a fashion as possible. The syllabus states that this is “often also true of these documents in traditional lifecycles”. This is not our experience!

Testers will also produce test metrics from defect reports and test results logs, and again there is an emphasis on a lightweight approach.

2.1.3 Test Levels

Learning Objectives

FA-2.1.2 K2 Describe how development and testing activities are integrated in Agile projects.

In sequential lifecycle models, the test levels are often defined such that the exit criteria of one level are part of the entry criteria for the next level. In some iterative models, this rule does not apply as test levels overlap. Requirement specification, design specification, and other development activities may overlap with test levels. Examples of test levels are component test, integration test, system test, system integration test, and acceptance test.

Agile models

In some Agile lifecycles, overlap occurs because changes to requirements, design, and code can happen at any point in an iteration. While Scrum, in theory, disallows changes to the user stories after iteration planning, in practice such changes sometimes occur. During an iteration, any given user story will typically progress sequentially through the following test activities:

- Unit testing

  Typically done by the developer.

- Feature acceptance testing

  This is sometimes broken into two activities:

  - Feature verification testing,

    Which is often automated, may be done by developers or testers, and involves testing against the user story’s acceptance criteria

  - Feature validation testing

    Which is usually manual and can involve developers, testers, and business stakeholders working collaboratively to determine whether the feature is fit for use, to improve visibility of the progress made, and to receive real feedback from the business stakeholders

In addition, there is often a parallel process of regression testing occurring throughout the iteration. This involves re-running the automated unit tests and feature verification tests from the current iteration and previous iterations, usually via the continuous integration framework.

In some Agile teams, there may also be a system test level which starts once the first user story is ready for such testing. This can involve executing functional testing, as well as non-functional tests for performance, reliability, usability, and other relevant quality characteristics.

Acceptance testing

Agile teams can employ various forms of acceptance testing (using the term as explained in the Foundation Level syllabus [ISTQB_FL_SYL].

- Internal alpha tests and external beta tests.
- User acceptance tests, operational acceptance tests, regulatory acceptance tests, and contract acceptance tests.

Either of these types of acceptance testing may be undertaken at the close of each iteration, after the completion of each iteration, or after a series of iterations.

2.1.4 Testing and Configuration Management

Agile projects typically make much use of automated tools to support the development and testing processes. This is not to suggest that traditional projects do not use tools extensively but with Agile projects tool support is particularly crucial to success. Frameworks that allow the continuous integration of new and changed software with the system work together with testing and configuration management tools that automate the build and testing process.

Developers use tools for static analysis, unit testing and code coverage. Developers frequently check the code and unit tests into a configuration management system that runs static analysis on all checked-in code and then uses automated build and test frameworks to compile the code and run the unit tests measuring the code coverage achieved.

See [Kubaczkowski] for a case study.

Regression risk

Automated testing and build tools help to manage the regression risk associated with the frequent change that often occurs in Agile projects. However, over-reliance on automated unit testing alone to manage these risks can be a problem, as unit testing often has limited defect detection effectiveness [Jones11]. Automated tests at the integration and system levels are also required.

Functional tests at the integration and system levels may be run as part of the continuous integration framework. These tests may be created using functional testing harnesses, open-source user interface functional test tools, or commercial tools. In some cases, due to the duration of the functional tests, some or all of the functional tests are separated from the unit tests and run less frequently. For example, unit tests may be run each time new software is checked in, while the longer functional tests are run only every few days.

Confirming the build

One goal of the automated tests is to confirm that the build is functioning and installable. If any automated test fails, the team should fix the underlying defect in time for the next code check-in. This requires an investment in real-time test reporting to provide good visibility into test results. This approach helps reduce expensive and inefficient cycles of “build-install-fail-rebuild-reinstall” that can occur in many traditional projects, since changes that break the build or cause software to fail to install are detected quickly.

2.1.5 Organisational Options for Independent Testing

Learning Objectives

FA-2.1.3 K2 Describe the role of independent testing in Agile projects.

Note that the syllabus makes the following statement: “As discussed in the Foundation Level syllabus [ISTQB_FL_SYL] independent testers are often more effective at finding defects.” This is not correct. The Foundation syllabus does make any such statement, thankfully, because it is not true.

Independent testers typically find some different defects from those found by testers more familiar with the software. It is therefore good practice to include some independent testers when possible to increase the overall effectiveness of the testing (i.e. to find more defects). We cannot say that one group of testers will be more effective than another.

Remember also that it is who designs the tests, not so much who runs them, that matters as far as independence is concerned.

Developer testing

In some Agile teams, developers create many of the tests in the form of automated tests. This will make the test verification less error prone, assuming that the automated verification is implemented correctly. However, it does not significantly change the degree of independence in the testing because it is the design of the test cases, not the execution of them, where the independence offers the greatest advantage. Besides, we need to retain the testing done by those familiar with the system.

Embedded testers

One or more full-time testers may be embedded within the team, performing many of the testing tasks. However, given those testers' position within the team, there is a risk of loss of independence and objective evaluation.

On-demand independent testers

Other Agile teams retain fully independent, separate test teams, and assign testers on-demand during the final days of each iteration. This can preserve independence, allowing these testers, in theory at least, to provide a more objective and unbiased evaluation of the software. However, time pressures, lack of understanding of the new features in the product, and relationship issues with business stakeholders and developers often lead to problems with this approach.

Separate test team

A third option is to have an independent, separate test team where testers are assigned to Agile teams on a long-term basis, at the beginning of the project, allowing them to maintain their independence while gaining a good understanding of the product and strong relationships with other team members. In addition, the independent test team can have specialised testers outside of the Agile teams to work on long-term and / or iteration-independent activities, such as developing automated test tools, carrying out non-functional testing, creating and supporting test environments and data, and carrying out test levels that might not fit well within a sprint (e.g. system integration testing).

2.2 Status of Testing in Agile Projects

Learning Objectives

FA-2.2.1	K2	Describe the tools and techniques used to communicate the status of testing in an Agile project, including test progress and product quality.
FA-2.2.2	K2	Describe the process of evolving tests across multiple iterations and explain why test automation is important to manage regression risk in Agile projects.

Change happens rapidly in Agile projects. This change means that test status, test progress, and product quality constantly evolve, and testers must devise ways to get that information to the team.

It is important that the whole team has visibility of the current status of the project. They will want to see which user stories have been implemented, how much work remains to be done on each of the remaining stories. This visibility is important to help the team make decisions that help them stay on track to the successful completion of each iteration and avoid unforeseen surprises.

In addition, change can affect existing features from previous iterations. Therefore, manual and automated tests must be updated to deal effectively with regression risk.

2.2.1 Communicating Test Status, Progress, and Product Quality

Learning Objectives

FA-2.2.1 K2 Describe the tools and techniques used to communicate the status of testing in an Agile project, including test progress and product quality.

Agile teams progress by having working software at the end of each iteration. Since each iteration is of a fixed duration, the team will need to monitor the progress of all work items in the iteration to ensure that the goal of working software is achieved by the end of the iteration.

Different methods

Testers in Agile teams use one or more of a variety of methods to record the current status and test progress, including:

- test automation results,
- progression of test tasks and stories on the Agile task board, and
- burndown charts showing the team's progress.

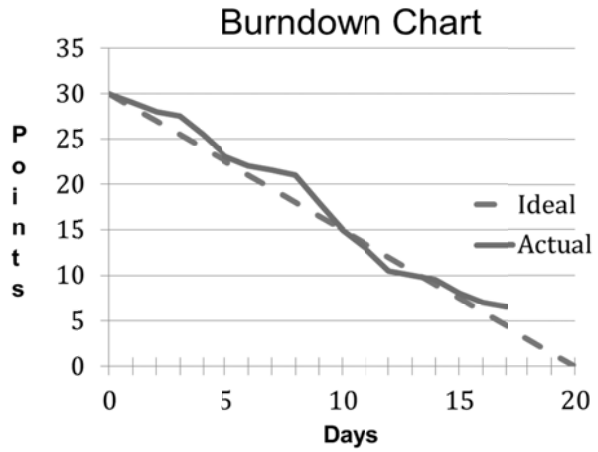
This information can then be communicated to the rest of the team using media such as wiki dashboards and dashboard-style emails, as well as verbally during stand-up meetings.

Test results (automated or manual) are one of the best ways to measure progress. Test results in terms of the number of tests created, run and passed may be displayed on large charts for all to see.

Tools

Agile teams may use tools that automatically generate status reports based on test results and task progress, which in turn update wiki style dashboards and emails. This method of communication may also gather additional metrics from the testing process that can be used in process improvement. Communicating test process status in such an automated manner frees testers' time for more value adding work such as focusing on the design and execution of additional test cases.

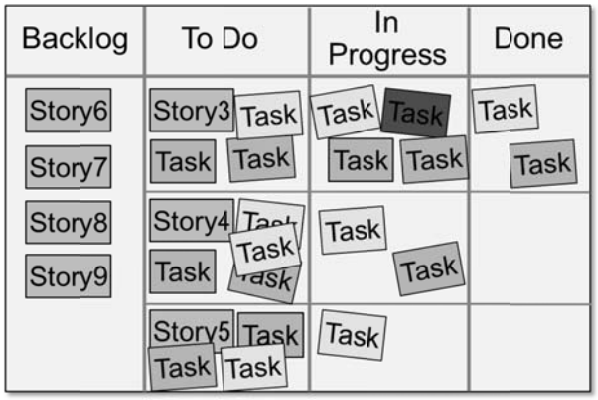
Teams may use burndown charts to track progress across the entire release and/or within each iteration. A burndown chart [Crispin08] represents the amount of work left to be done (based on the estimate assigned to each task by the team during the release and iteration planning meetings) against the time allocate to the release or iteration. The chart below shows a burndown chart for an iteration that is on day 17 of 20 allocated for it. The solid line shows the actual points implemented and the dashed line shows the planned rate of implementation.



Agile task boards

To provide an instant, detailed visual representation of the whole team's current status, including the status of testing, teams may use Agile task boards. The story cards, development tasks, test tasks, and other tasks created during iteration planning (see Section 1.2.5) are written on cards (or sticky notes) which, together with the story cards are stuck on the task board. The cards are typically colour-coordinated so it is easy to spot the type of task without having to read it. For example, yellow for development tasks, green for testing tasks and red for defects. Colour-coding also helps the team make a quick assessment of the current status with a quick glance at the task board. Task boards are often white boards, cork boards, magnetic sheets or just an area of the office wall that is easily visible to the whole team.

There is no standard for task boards. How they are organised and laid out is decided by the team. However, it is common to have the task board is organised into columns, one for each stage of task progress such as 'to do', 'work in progress', 'verify', and 'done'. During the iteration, progress is represented by the movement of the tasks across the task board. Also, having separate rows marked out on the board for each user story that is in progress or is about to be started.



As an alternative to a physical task board Agile teams may use software tools to maintain their story cards and virtual Agile task boards. The tools can automate dashboards and status updates.

Tasks on the task board

Testing tasks on the task board relate to the acceptance criteria defined for the user stories. When the automated tests, manual scripted tests, and/or exploratory tests for a test task achieve a passing status, the task moves into the done column of the task board.

The whole team reviews the status of the task board regularly, often during the daily stand-up meetings, to ensure tasks are moving across the board at an acceptable rate. If any tasks (including testing tasks) are not moving or are moving too slowly, the team reviews and addresses any issues that may be blocking the progress of the those tasks.

Daily stand-up meeting

The daily stand-up meeting includes all members of the Agile team. At this meeting, each team member communicates the current status of their work. The agenda for each member is [Agile Alliance Guide]:

- What have you completed since the last meeting?
- What do you plan to complete by the next meeting?
- What is getting in your way?

Any issues that may block test progress are communicated during the daily stand-up meetings, so the whole team is aware of the issues and can resolve them accordingly.

Customer satisfaction surveys

To improve the overall product quality, many Agile teams perform customer satisfaction surveys to receive feedback on whether the product meets customer expectations.

Other metrics

Teams may use other metrics similar to those captured in traditional development methods, such as test pass / fail rates, defect discovery rates, confirmation and regression test results, defect density, defects found and fixed, requirements coverage, risk coverage, code coverage, and code churn.

As with traditional development methods these metrics are used to help identify opportunities for improving the product quality. The metrics captured and reported should be relevant and aid decision-making. Metrics should not be used to reward, punish, or isolate any team members.

2.2.2 Managing Regression Risk with Evolving Manual and Automated Test Cases

Learning Objectives

FA-2.2.2 K2 Describe the process of evolving tests across multiple iterations and explain why test automation is important to manage regression risk in Agile projects.

In an Agile project, as each iteration completes, the product grows. Therefore, the scope of testing also increases with each iteration. Along with testing the code changes made in the current iteration, testers also need to verify that the features implemented in previous iterations have not been adversely affected by the latest changes. This is achieved by regression testing the software product during every iteration, a task that grows with each iteration.

Automated regression testing is an important part of the Agile tester's armoury. No Agile team will succeed by doing only manual testing [Crispin08].

When a test fails

Automated tests contained in the regression test set are generally run as part of the daily continuous build in the continuous integration environment, and again when a new build is deployed into the test environment.

As soon as an automated regression test fails, the team stops and investigates the reasons for the failing test. The test may have failed due to legitimate functional changes in the current iteration, in which case the test and / or user story may need to be updated to reflect the new acceptance criteria. Alternatively, the test may need to be removed (or retired/archived) if another test has been built to cover the changes. However, if the test failed due to a defect, it is good practice for the team to fix the defect prior to progressing with new features. On occasions it may be necessary to delay fixing the defect until the next iteration (or even later) if it has a low impact on the customer.

Code churn

The risk of introducing regression in Agile development is high due to extensive code churn (the number of lines of code added, modified, or deleted from one version to another). Since responding to change is a key Agile value, changes to previously delivered features are not only possible but likely in many situations to better meet business needs and fix defects. Furthermore, code that implements existing features may be refactored to deal with other forms of technical debt, for example, to improve its maintainability, testability or reusability without affecting the features that it implements.

The ability to effectively and efficiently check that code changes have not regressed the system (introduced defects) is vital to the success of Agile projects. The greater the code churn, the more comprehensive the regression testing needs to be.

Test asset maintenance

In order for the team to maintain velocity without incurring a large amount of technical debt, it is critical that teams invest in test automation at all test levels as early as possible. It is also critical that all test assets such as automated tests, manual test cases, test data, and other testing artefacts are kept up-to-date with each iteration. Technical debt applies to these other software artefacts as well, not just to the code that implements the product.

It is critical that testers have the ability to quickly identify and update test cases from previous iterations and / or releases that are affected by the changes made in the current iteration. Defining how the team designs, writes, and stores test cases should occur during release planning. Good practices for test design and implementation need to be adopted early and applied consistently. The shorter timeframes for testing and the constant change in each iteration will increase the impact of poor test design and implementation practices.

It is highly recommended that all test assets be maintained in a configuration management system in order to enable version control, to ensure ease of access by all team members, and to support making changes as required due to changing functionality while still preserving the historic information of the test assets. A good configuration management system can help the team identify dependencies between the software artefacts ahead of running tests, thereby saving some effort in dealing with test failures due to missed dependencies.

Use of test automation, at all test levels, allows Agile teams to provide rapid feedback on product quality. Well-written automated tests provide a living document of system functionality [Crispin08]. By checking the automated tests and their corresponding test results into the configuration management system, aligned with the versioning of the product builds, Agile teams can review the functionality tested and the test results for any given build at any given point in time.

Reviewing test cases

Because complete repetition of all tests is seldom possible, especially in tight-timeline Agile projects, testers need to allocate time in each iteration to review manual and automated test cases from previous and current iterations to select test cases that may be candidates for the regression test suite and to retire test cases that are no longer relevant.

Tests written in earlier iterations to verify specific features may have little value in later iterations due to feature changes or new features which alter the way those earlier features behave. Also, later tests may supersede earlier tests in terms of the value they provide. Removing redundant tests will avoid wasted effort running and maintaining them.

While reviewing test cases, testers should also consider suitability for automation. The team needs to automate as many of the tests from previous and current iterations as will give value. This allows automated regression tests to reduce regression risk with less effort than manual regression testing would require. The reduced regression test effort frees the testers to undertake more value added work such as more thoroughly testing new features and functions in the current iteration.

Testing before source code is checked in

Static analysis and the automated unit tests should be run by the developers before they check the source code into the mainline of the configuration management system. This will ensure that the code changes do not break the software build and cause delays to the availability of the next build (which may affect the whole team). To reduce build breaks, which can slow down the progress of the whole team, code should not be checked in unless static analysis reports the code to be clear of all statically detectable defects and anomalies and all automated unit tests pass. Static analysis and automated unit test results provide immediate feedback on code and build quality, but not on product quality.

Build verification tests

Automated tests can be run continuously against the system. An initial subset of automated tests to cover critical system functionality and integration points should be run immediately after a new build is deployed into the test environment. These tests are commonly known as

build verification tests. Results from the build verification tests will provide instant feedback on the software after deployment, so teams don't waste time testing an unstable build.

Automated acceptance tests

Automated acceptance tests are run regularly as part of the continuous integration full system build. These tests are run against a complete system build at least daily, but are generally not run with each code check-in as they take longer to run than automated unit tests and would otherwise slow down code check-ins and reduce the frequency of new builds.

The test results from automated acceptance tests provide feedback on product quality with respect to regression since the last build, but they do not provide status of overall product quality.

Other testing tasks to automate

In addition to test execution automation, other testing tasks may also be automated including:

- test data generation;
- loading test data into systems;
- deployment of builds into the test environments;
- restoration of a test environment (e.g., database or website data files) to a baseline;
- comparison of actual with expected data outputs.

Automation of these tasks reduces the overhead of regression testing, avoids possible human errors and allows the team to spend more time developing and testing new features.

2.3 Role and Skills of a Tester in an Agile Team

Learning Objectives

FA-2.3.1 K2 Understand the skills (people, domain, and testing) of a tester in an Agile team.

FA-2.3.2 K2 Understand the role of a tester within an Agile team.

In accordance with the Agile principle 'work together daily', testers must closely collaborate with all other team members and with the business stakeholders outside the team. This has a number of implications in terms of the skills a tester must have and the activities they perform within an Agile team. In this section we take a closer look at these skills and activities.

2.3.1 Agile Tester Skills

Learning Objectives

FA-2.3.1 K2 Understand the skills (people, domain, and testing) of a tester in an Agile team.

Skills for testers in general

Testers are testers whether or not they work in Agile development projects. There are many important skills that it is beneficial for testers to have regardless of their project type. These skills include any that help testers undertake the following core testing activities:

- analyse test basis documents such as requirements specifications, use cases, design specifications, and in some situations, code;
- design good test cases, including the use of testing techniques, the design of test data and defining expected results;
- execute tests, verify the results and record the whole activity to the required level of detail;
- report defects accurately and objectively;
- participate in product risk management activities.

In addition, test professionals should be well-organized, attentive to detail and have good written and verbal communication skills. Testers will benefit from interpersonal skills such as how to give and receive criticism, influencing and negotiating, and working effectively with others.

It is not necessary for any one tester to possess all of these skills in order to make a valuable contribution to a testing effort. However, it would be helpful to have each of these skills represented within a test team.

Note that the Agile syllabus states that 'Agile testers should have all the skills mentioned in the Foundation Level syllabus [ISTQB_FL_SYL]'. However, there are only two skill sets specifically stated in the syllabus: interpersonal skills and test automation skills. In the list above can be reasonably inferred from the syllabus and, we hope, common sense!

Competencies for Agile testers

The Agile syllabus states that in addition to the skills outlined above, "a tester in an Agile team should be competent in test automation, test-driven development, acceptance test-driven development, white-box, black-box, and experience-based testing". Wow! This is some list. We believe it fairer to say that a tester in an Agile team will benefit from having a number of these competencies. Much of the Agile literature describes the ideal Agile team member as a generalist, being able to undertake any task. In many situations this is unlikely to be practical though the aim of sharing knowledge and working together should remain.

Skills of particular importance to Agile testers

As Agile methodologies depend heavily on collaboration, communication, and interaction between the team members as well as stakeholders outside the team, testers in an Agile team will have a greater need for good interpersonal skills.

The following skills are common requirements for all testers but they have particular importance for testers in Agile teams:

- be positive and solution-oriented with team members and stakeholders;
- display critical, quality-oriented, sceptical thinking about the product;
- accurately evaluate and report test results, test progress, and product quality.

Skills more specific to Agile testers

The following skills are more specific to testers in Agile teams:

- actively acquire information from stakeholders,

  rather than relying on written specifications;
- work effectively to define testable user stories,

  especially acceptance criteria, with customer representatives and stakeholders;
- collaborate within the team,

  working in pairs with programmers and other team members;
- respond to change quickly,

  including changing, adding or improving test cases;
- plan and organise their own work.

Continuous skills growth, including interpersonal skills growth, is essential for all testers, including those on Agile teams.

2.3.2 The Role of a Tester in an Agile Team

Learning Objectives

FA-2.3.2 K2 Understand the role of a tester within an Agile team.

Testing tasks

The role of a tester in an Agile team includes activities that generate and provide feedback not only on test status, test progress, and product quality, but also on process quality.

In addition to the activities described elsewhere in this syllabus, these activities include:

- understanding, implementing, and updating the **test strategy**;
- measuring and reporting **test coverage** across all applicable coverage dimensions;
- ensuring proper use of **testing tools**;
- configuring, using, and managing **test environments** and test data;
- reporting **defects** and working with the team to resolve them;
- **coaching** other team members in relevant aspects of testing;
- ensuring the appropriate **testing tasks** are scheduled during release and iteration planning;
- actively collaborating with developers and business stakeholders **to clarify requirements**, especially in terms of testability, consistency, and completeness;
- participating proactively in **team retrospectives**, suggesting and implementing improvements.

Who does the testing?

Within an Agile team, each team member is responsible for product quality and plays a role in performing test-related tasks.

Test-related organisational risks

Agile projects may encounter some test-related organisational risks:

1. because testers work so closely with developers
   they may lose the appropriate tester mind-set;
2. testers may become tolerant of or silent about
   inefficient, ineffective, or low-quality practices within the team;
3. testers may not keep pace with the incoming changes
   in time-constrained iterations.

To mitigate these risks, organisations may consider variations for preserving independence discussed in Section 2.1.5.

2.4 Summary

Testing and development activities

Testing and development activities are structured differently and are described by one or more of three software development models: sequential, iterative and incremental. Agile is both iterative and incremental. Agile approaches can be implemented in different ways to adapt to the context of a given project. These differences include names used, project work products produced and the use of independent testing.

The main differences between traditional development lifecycles and Agile approaches include:

- short iterations (typically 1 to 4 weeks maximum)

- working software (each iteration produces working software)
- collaborative approach (all team member are involved with testing and testers can be involved in non-testing tasks)
- minimise technical debt (principle of continuous attention to technical excellence and good design to enhance agility)
- pairing (cross-discipline paired sessions of 30 to 90 minutes)
- sharing knowledge (testers coach other team members)
- test automation (critical to the success of Agile)
- embracing change (lightweight work product documentation)

Project work products

There are three categories of work product of interest to testers:

- business-oriented (e.g. user stories and acceptance criteria)
- development (e.g. code, automated unit tests)
- test (e.g. manual and automated tests, test dashboards)

Agile avoids detailed documentation that doesn't deliver value to the customer.

Test levels

With traditional software lifecycles there may be entry and exit criteria that govern when one test level ends and the next test level can start. There is often no clear distinction between different test levels within an Agile iteration, although testing at different levels does occur. However, acceptance testing may be undertaken at the close of each iteration or after a series of iterations.

Configuration Management

Every time a developer checks-in code to the configuration management system, static analysis, automated unit tests and code coverage measurement is automatically performed. The system build process is run frequently (at least daily) which is followed by automated build verification tests, integration tests and possibly system tests.

Independent testing

The use of independence in testing can enhance the effectiveness of testing and may take the form of automated unit tests, testers embedded within the team, on-demand independent testers and a separate test team.

Communicating test status

Because of the frequency of change within an Agile project, it is important for the whole team to have visibility of the current status of the project. This is typically achieved through a number of methods including the results of frequent automated test runs, the use of a task board and burndown charts. Daily stand-up meetings keep the whole team informed of progress and any obstacles to progress. Customer satisfaction surveys can be used to gain feedback on the overall quality of the product. Several other metrics may be used, including many used by projects using traditional lifecycles.

Managing regression risk

Frequent builds and automated test runs help mitigate regression risk. Whenever an automated regression test fails, the team investigates the reasons and fixes the issue. Code churn (the number of lines of code added, changed or removed) gives an indication of regression risk.

Other mitigation activities include the maintenance of test assets (e.g. to remove redundant tests), reviewing test cases (to determine which should be automated and/or run) and the automation of testing tasks in addition to test execution.

Agile tester skills

Many skills are beneficial for testers regardless of the software development lifecycle being followed. This includes analysis, attention to detail, problem solving, domain knowledge, diligence, written and verbal communication skills, interpersonal skills and test automation skills.

Agile testers can benefit from additional competencies and skills, including test automation, test-driven development, acceptance test-driven development, white-box, black-box and experience-based testing.

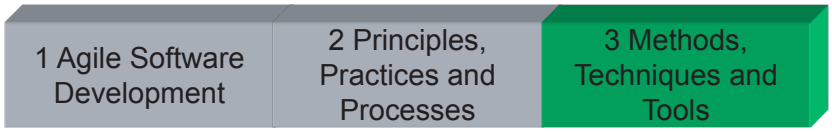
Agile tester roles

Testing tasks that are particularly relevant to Agile testers include measuring and reporting test coverage, use of tools, configuring, using and maintaining test environments, coaching testing, clarifying requirements and participating in team retrospectives.

ISTQB Foundation Extension Agile Tester

Module 3:
Agile Testing Methods,
Techniques and Tools

Agile Testing Methods, Techniques and Tools



Contents

Testing methods

appropriate for any development project (Agile or not) …

write tests first

… but applied early in Agile projects

right tests, right time, right test level

early defect prevention, detection & removal

Agile testers have a key role in guiding the use of these practices

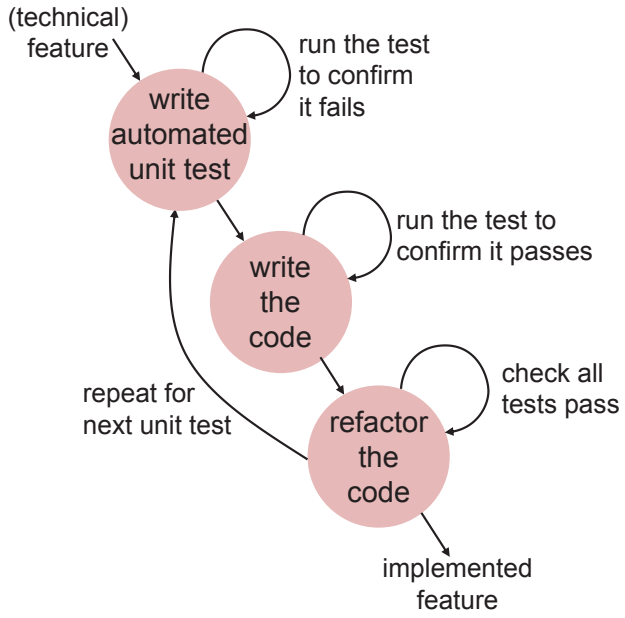
4

Contents

5

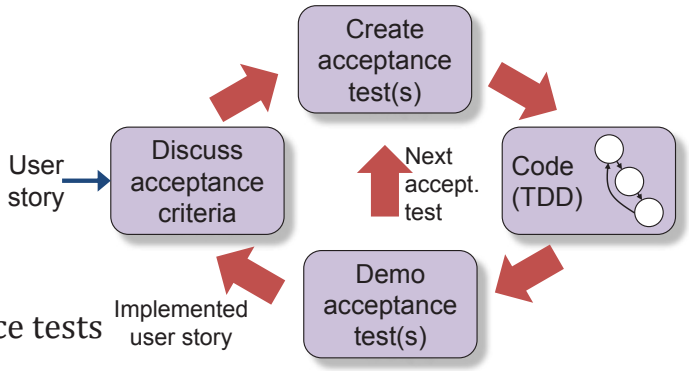
Test-driven development (TDD)

- write an automated unit test
  - confirm it fails
- write the code
  - one test at a time
  - confirm test passes
- refactor the code
  - to ensure quality
- all tests run again
  - to prevent regression
- repeat for next unit test



Simple ATDD cycle

- complementary to TDD
- collaborative approach
  - involves whole team
- customer identifies acceptance criteria for a user story
  - team discussion
- team identify acceptance tests
  - automates tests
  - implements code
    - ► to pass tests
- demo acceptance tests



Behaviour-Driven Development (BDD)

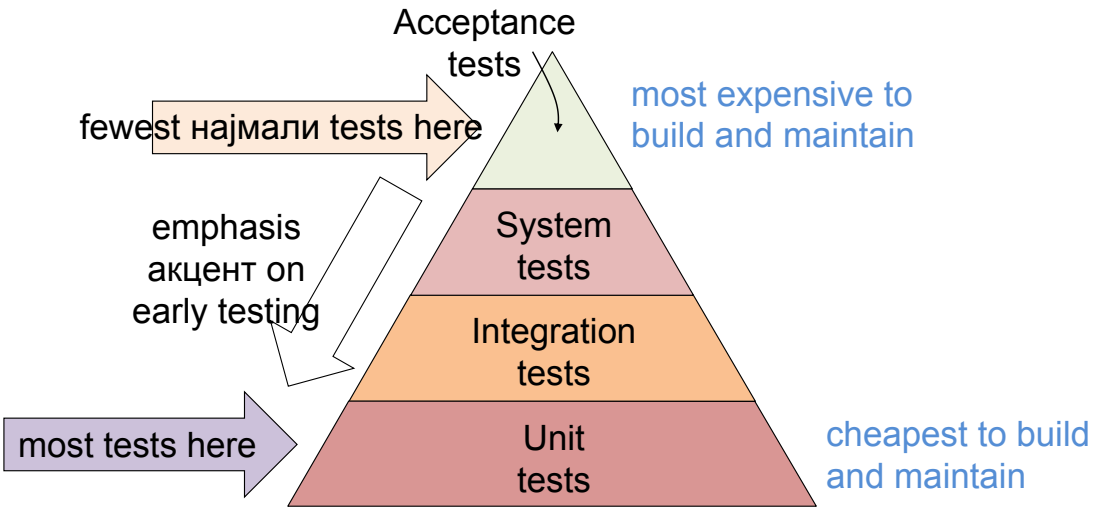
- similar to ATDD
  - but with focus on behaviour
- uses template for acceptance criteria:
  - **Given** some initial context
  - **When** an event occurs
  - **Then** ensure some outcome
- e.g.
  - **Given** I have a discount code,
  - **When** I enter it on checkout,
  - **Then** the purchase price will be discounted.
- applies to all levels
  - unit tests are more robust
    - ► not affected by refactoring
- specific tool support
  - BDD framework
    - ► generates code to help with creating automated tests
    - ► e.g. Rspec and Cucumber

8

Contents

9

Test Pyramid



10

Characteristics of the test levels

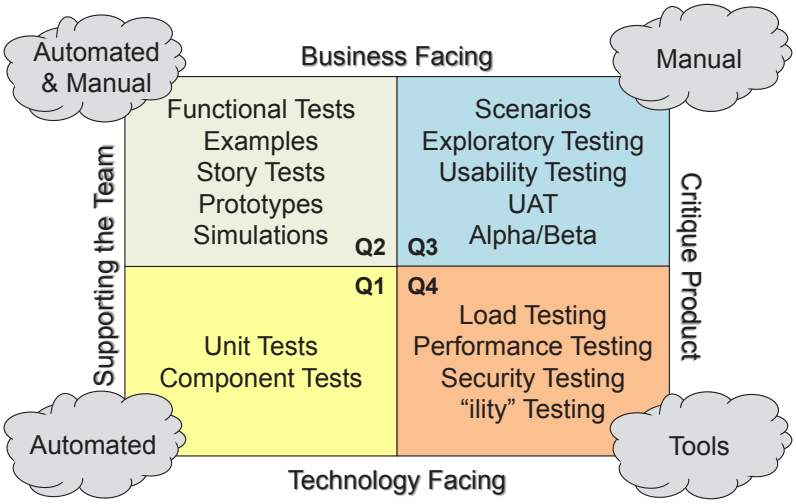
Acceptance tests	longest to run	some run every build, others daily	smallest ROI	domain specific test language (customer readable)	GUI-based tools
System tests	run daily, or less frequently				
Integration tests		run with every build		may not be customer readable	API-based tools & unit test frameworks
Unit tests	shortest to run	run with every build	largest ROI	not customer readable	

11

Contents

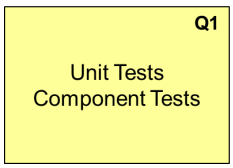
12

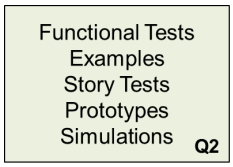
Agile testing quadrants



13

Quadrants 1 and 2





Q1

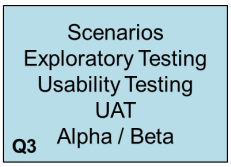
- unit and component level
  - technology facing tests
  - support developers
  - verify functionality of code
- automated
  - included in continuous integration process
    - ► run on every build

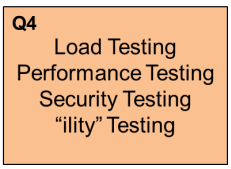
Q2

- system level
  - business facing tests
  - support team (higher level)
  - confirm product behaviour
    - ► check acceptance criteria
- mostly automated, some manual
- executable part of specification
  - business representatives use and help write them

14

Quadrants 3 and 4





Q3

- system and user acceptance level
  - business facing
  - critique the product
    - ► using realistic scenarios/data
- manual
  - by end users
    - ► usability and UAT
  - by testers
    - ► exploratory

Q4

- system and operational acceptance level
  - technology facing tests
  - critique the product
    - ► check quality characteristics
- supported by tools
  - much manual intervention

15

Contents

16

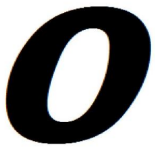
Team work

- cross-functional — sharing testing skills
- self-organising — team identifies the roles
- co-located — easy access and common visibility of progress
- collaborative — testers, developers and business representatives
- empowered — problem solving together
- committed — questions and evaluate product's behaviour
- transparent — progress visible (task board)
- credible — tester ensures credibility of test strategy
- open to feedback — retrospectives: lean from successes and failures
- resilient — respond to change

17

Sprint zero and integration

Sprint zero



- scope
- initial architecture
- tool support
- test strategy
- initial quality risk analysis
- test metrics
- define 'done'
- create task board
- test completion criteria

Integration

- aim to deliver value with every sprint
  - integration critical
    - ► test automation vital
  - testing strategy key
    - ► which tests
    - ► which test type
    - ► when executed
  - understand dependencies

18

Test planning and Agile testing practices

Test planning

- starts at iteration zero
  - updated each sprint
- addresses issues:
  - scope
  - responsibilities
  - environment and data
  - timing/sequencing/ dependencies/prerequisites
  - risk assessment/mitigation
- sprint planning outcome:
  - testing tasks on task board

Agile testing practices

- pair testing
  - two people work together on one task
- incremental design
  - start with simple tests
    - ► progress to more complex
- mind mapping
  - use mind maps to capture ideas and other information
    - ► e.g. test strategies, test data

19

Contents

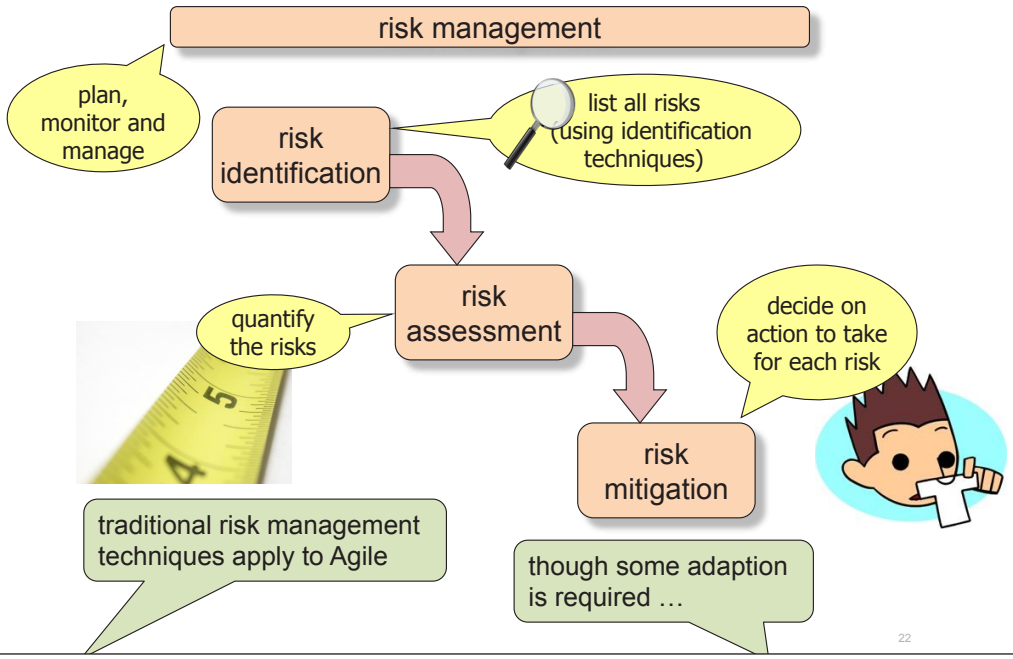
20

A challenge of testing

- an objective of testing
  - find as many defects prior to release
    - ► so as few as possible are found after release
  - within time and cost constraints
- challenge is the proper соодветен:
  - selection of tests
  - allocation of effort
  - prioritization of tests
- we can use risk to help solve this problem
  - risk-based testing

21

Managing quality risks



22

Contents

23

What is risk?

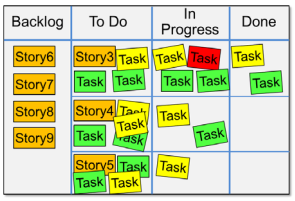
- risk is the possibility of an undesired outcome
  - if it occurs it will decrease stakeholders perceptions of product quality or project success
- two types of risk
  - product risk (quality risks)
    - ► primary effect of the potential problem is on product quality
  - project risk (planning risks)
    - ► primary effect of the potential problem relates to project success
- the level of risk depends on two factors:
  - likelihood of the problem occurring
  - impact of the problem should it occur

} risk level

24

Identifying and assessing risk

- when?
  - release planning
  - iteration planning
- outcome of planning
  - tasks on task board

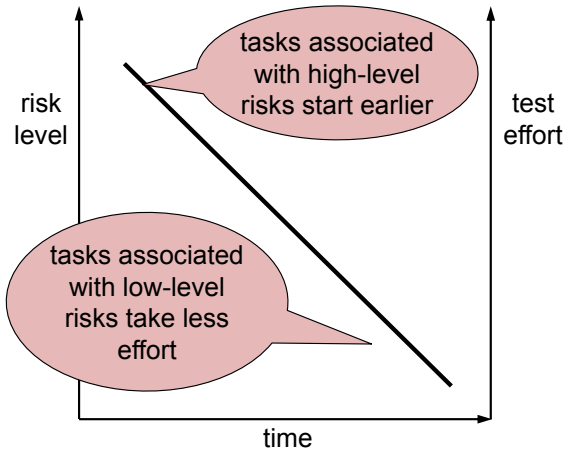


- tasks prioritised by risk

- quality risk examples
  - users may not be able to pay by credit card
    - ► functional risk
  - system may not sustain 20 concurrent users
    - ► efficiency risk
- project risk examples
  - delivery of hardware for test environment may be late
  - a team member may resign

25

Prioritising tasks based on risk



26

Example risk analysis process (iteration planning)

1. gather team together
2. list iteration backlog items
3. identify quality risks for each backlog item
4. assess each identified risk
   - categorize
   - determine risk level
5. determine extent of testing for each risk level
6. determine appropriate mitigation tasks
   - based on category and level of risk

consider all relevant quality characteristics

based on impact & probability

e.g. coverage, effort, number of test conditions

e.g. test techniques, types of testing

27

Risk re-evaluation and static testing

Risk re-evaluation

- risks change throughout project
  - re-evaluate existing risks
  - identify new risks
- evaluate effectiveness of risk mitigation tasks

Static testing

- can mitigate risks before test execution
  - e.g. problems with user stories
    - ► more thorough review

28

Contents

29

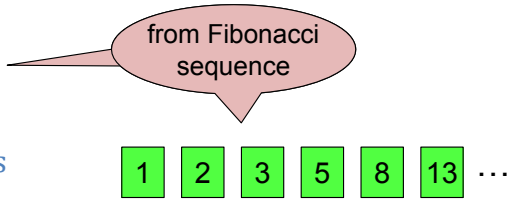
Story points

- estimate 'story points'
  - an indication of the work required to implement a story
    - ► story A: 6 points
    - ► story B: 3 points
      - – story A will take twice the work taken by story B
- 1 story point =
  - 1 effort day
  - 1 paired day
  - 1 effort week
  - or other …
- story points are team specific
  - one teams points are different to those of other teams
- team's velocity:
  - number of story points implemented in one iteration

30

Planning poker

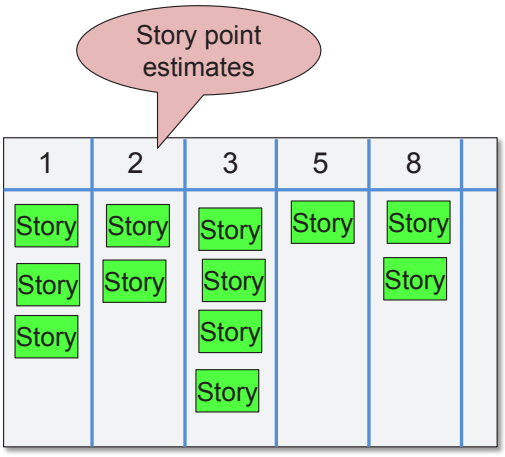
- common estimation technique
  - product owner reads story
  - estimators as questions
    - ► consider
      - – coding and testing effort
      - – story complexity and risk level
      - – scope of testing
  - each estimator selects a card
    - ► all revealed at same time
  - if all the same: got it!
  - otherwise discuss differences
    - ► then repeat cycle
- product owner does <u>not</u> estimate
  - or argue with estimates made
- user story estimates
  - are owned by the team



31

Triangulation

- check estimation consistency
- helps prevent 'drifting'
  - where estimates become progressively more pessimistic/optimistic
- group stories according to estimates
- check new estimates as stories are added, e.g.
  - are all 3's about the same?
  - are 2's less than half the 5's?
  - are 5's more than half the 8's?



32

Contents

33

Requirements

- requirements defined by
  - user stories
  - acceptance criteria and tests
  - conversations
    - ► including conversation notes
- tests are based on requirements

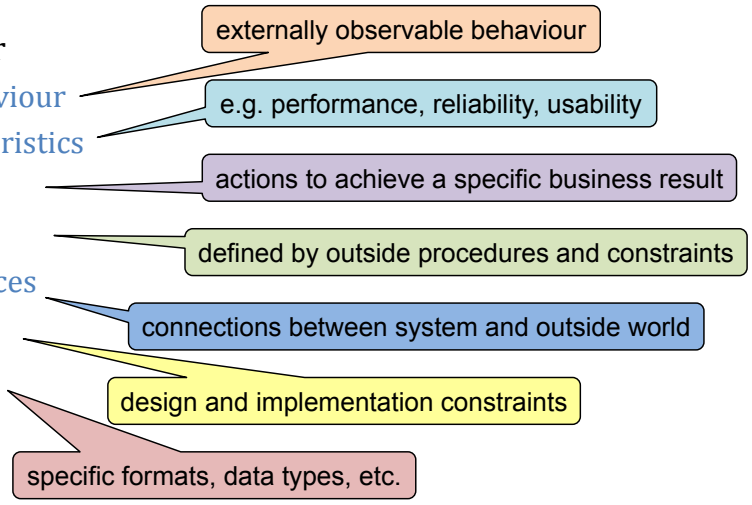
Requirement =
Story + Tests + Conversation

- other sources of information
  - experience
  - existing system
  - code, architecture, design
  - user profiles
  - quality risks
  - defects
  - defect taxonomy
  - applicable standards
- the acceptance tests help to validate the code

34

Acceptance criteria

- topics to consider
  - functional behaviour
  - quality characteristics
  - scenarios
  - business rules
  - external interfaces
  - constraints
  - data definitions



35

Definitions of 'done'

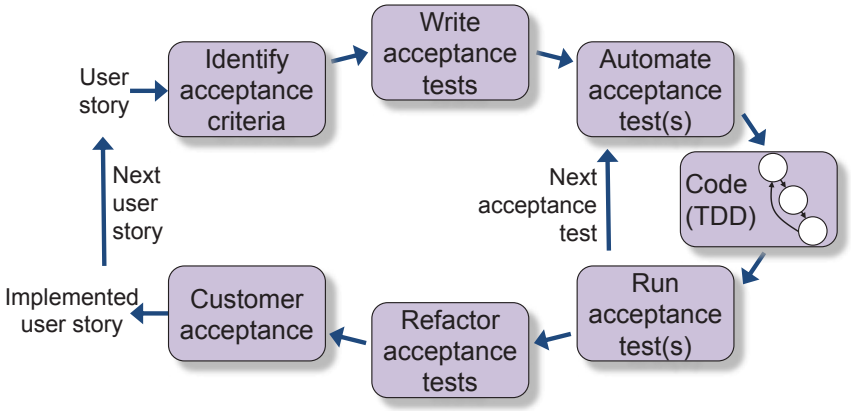
- test levels
  - unit testing
    - ► static analysis: 0 defects
    - ► 100% decision coverage
  - integration testing
    - ► unit interfaces tested
    - ► all defects found are reported
  - system testing
    - ► regression tests automated
    - ► no unresolved major defects
- user story
  - testable acceptance criteria
  - all tasks estimated by team
- feature
  - all user stories user approved
  - system tests all passed
- iteration
  - all features tested
  - outstanding defects logged
- release
  - test coverage
  - quality, time and cost

36

Contents

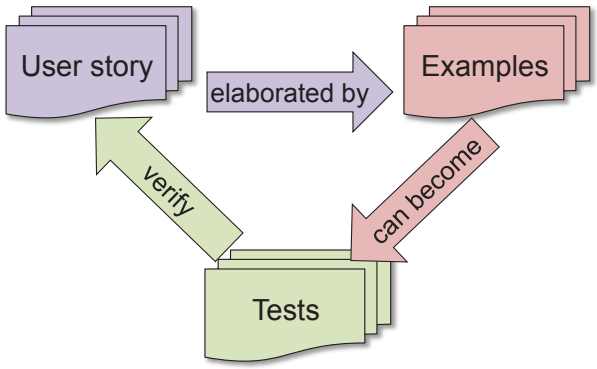
37

Detailed ATDD cycle



38

Requirements, examples and tests

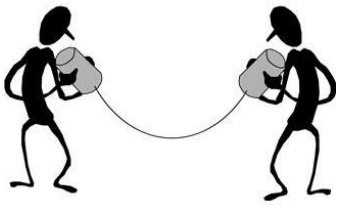


Source: Adapted from [Adzic09] referencing an original created at the 2nd Agile Alliance Functional Testing Tools Workshop, Toronto, 2008

39

More about ATDD

- acceptance tests extend description of user story
  - expressed in natural language
    - ► that stakeholders understand
  - "executable specification"
- primary purpose of acceptance tests:
  - to help team understand customer expectations
    - ► i.e. communication
- ATDD creates reusable tests for regression testing
  - automated or manual



40

Contents

41

Black-box test design

- testers and programmers
  - create tests based on user stories
  - acceptance criteria
- may use black-box test design techniques
  - for functional and some non-functional characteristics
- may use structure-based techniques
  - outside scope of syllabus
- equivalence partitioning
  - identifies sets of values that are handles in the same way
- boundary value analysis
  - identifies extreme values (min. and max. values)
- decision tables
  - good for complex business rules (explores combinations)
- state transition testing
  - transition between states (e.g. GUI navigation)

42

Contents

43

Exploratory testing

"An approach to software testing that emphasizes the personal freedom and responsibility of each tester to continually optimize the value of her work by treating learning, test design and test execution as mutually supportive activities that run in parallel throughout the project."

James Bach/Michael Bolton edit of a Cem Kaner suggestion.

- it's an approach
  - not a technique
- it's prepared
  - not unprepared
- it's a skill
  - that can be learnt

reacting with reasoned actions

beyond the obvious

- it involves
  - good design
  - critical thinking
  - careful observation
  - diverse ideas
  - rich resources

using appropriate methods

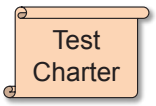
of software behaviour

tools, data, info. sources, techniques

44

Test charter and managing exploratory testing

Test charter



- guides the testing session
  - **purpose** – what to explore
  - **priority** – importance
  - **reference** – test basis
  - **setup** – testing prerequisites
  - **data** – required for testing
  - **actor** – intended user
  - **activities** – ideas for testing
  - **oracle** – how to evaluate
  - **variations** – alternatives

Managing exploratory testing

- sessions:
  - start with test charter
  - last 60 to 120 minutes
  - produce a session report
  - end with debrief
- session categories:
  - survey
  - analysis
  - deep coverage

45

Test quality and heuristics

Test quality

- depends (in part) on questions asked by tester
  - what is of most importance?
  - how may the system fail?
  - what happens if/when …?
  - are customer needs fulfilled?
- tester must use
  - creativity/intuition
  - skills/cognition
  - product/domain knowledge

Heuristics

- guidelines, checklists, rules of thumb
  - used to give test ideas
    - ► things to consider testing
- e.g. 'boundaries'
  - test on and just off boundary
- e.g. 'selection'
  - test none, some and all
- e.g. CRUD
  - create, read, update, delete

46

Tester notes

- record of testing session
  - made by tester during session
- possible content
  - test coverage
  - evaluation notes
  - risk/strategy list
  - issues
  - questions
  - anomalies
  - actual behaviour
- if recorded in standard format
  - tool support can aggregate data across multiple testers
  - or record in test/task management tool

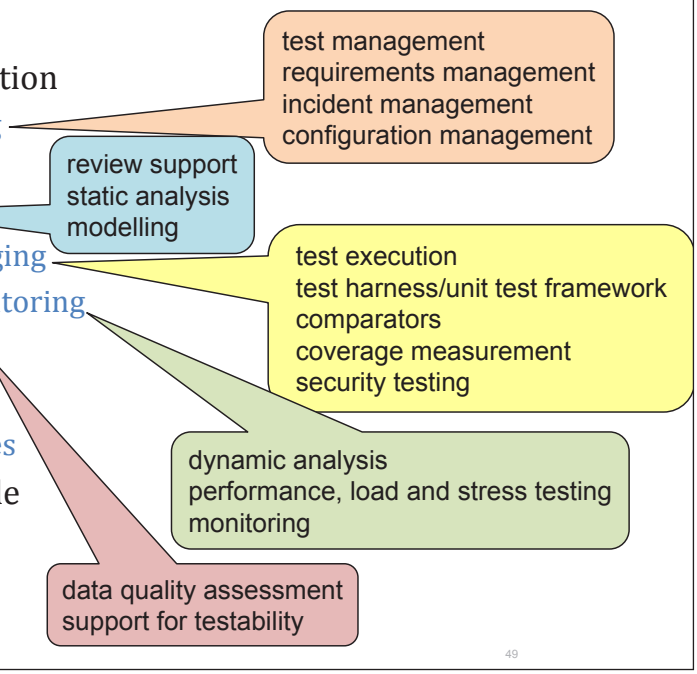
47

Contents

48

Tools in Agile projects

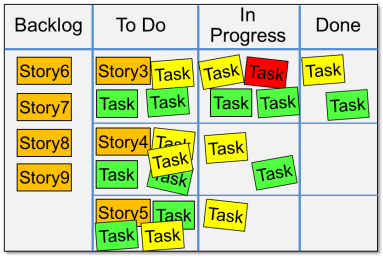
- tools covered at foundation
  - management of testing and tests
  - static testing
  - test execution and logging
  - performance and monitoring
  - specific testing needs
- applicable to Agile
  - though relevance varies
- other tools support Agile methods ...

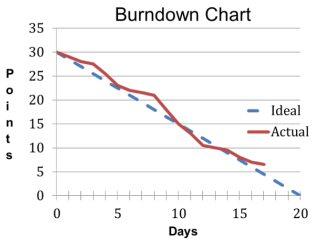


49

Task management and tracking tools

- may be used instead of whiteboard, corkboard
  - for story/task boards
- benefits of tools
  - stories/tasks don't get lost
    - ► link tasks with stories
  - capture estimates, show total
  - aggregate status updates
    - ► visible iteration/release status
  - visual representations for all
    - ► supports distributed teams
  - integrate with config. mgt.

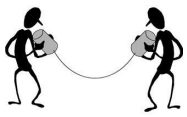




50

Communication and information sharing tools

- tools used <u>in addition</u> to e-mail, documents and verbal
  - 3 types of tool

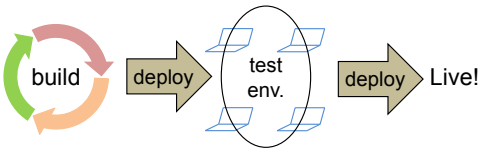


1. wiki
   - build & share knowledge base
     - ► features, discussions, prototype diagrams, photos
     - ► tools/techniques
     - ► metrics, charts, dashboards
     - ► conversations
2. instant messaging, teleconference (video/audio)
   - benefits
     - ► real-time direct comm.
     - ► distributed stand-up meetings
     - ► reduce telephone costs
3. desktop sharing & capture
   - benefits
     - ► distributed demos., reviews, pairing
     - ► capture demos.

51

Software build and distribution tools

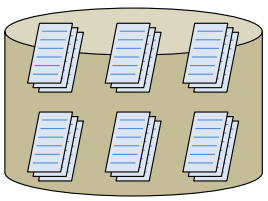
- used to implement continuous integration (see Section 1.2.4)
- benefits
  - earlier defect detection
  - regular/quick feedback
  - up-to-date software version
  - reduced regression risk
- risks
  - tool maintenance
  - continuous integration process implementation



52

Configuration management tools

- used for all testware artefacts
  - manual testing included
- benefits
  - traceability between software and testware versions
  - supports rapid change
  - prevents accidental changes
- distributed teams
  - require distributed configuration management tools



configuration management

53

Test design, implementation and execution tools

- test design
  - mind mapping tools
- test case management
  - part of task management?
- test data preparation
  - particularly useful when refactoring to reflect changes
- test data load
  - makes test environment setup efficient and reliable
- test execution
  - open source and commercial
  - support test-first approaches
    - ► BDD, ATDD, TDD
- exploratory testing
  - video capture and logging of sessions

54

Cloud computing and virtualization tools

- cloud computing
  - availability of large computer resources
    - ► less internal infrastructure
  - pay-as-you-go approach
    - ► flexibility
- virtualization tools
  - single server operates as though it is several
  - include snapshot capabilities
    - ► restore servers efficiently

55

Summary

- Agile testing methods
  - BDD, ATDD and TDD
    - ► test-first approaches
    - ► code written to pass tests
  - test pyramid
    - ► levels of testing
  - test quadrants
    - ► aligns test levels and types
- role of the tester
  - part of a team
    - ► collaboration
    - ► shared responsibilities
- assessing quality risks
  - risk-based to focus test effort
- estimating
  - planning poker, story points
- techniques in Agile projects
  - acceptance criteria
- tools in Agile projects
  - traditional tools apply
  - specific tools include:
    - ► task management, communication, build, configuration, test design to execution, virtualization

56

ISTQB Foundation Extension Agile Tester

Session 3:

Agile Testing Methods, Techniques and Tools

Terms
acceptance criteria, exploratory testing, performance testing, product risk, quality risk, regression testing, test approach, test charter, test-driven development, test estimation, test execution automation, test strategy, unit test framework.

From the ISTQB Glossary

acceptance criteria: The exit criteria that a component or system must satisfy in order to be accepted by a user, customer, or other authorized entity.

exploratory testing: An informal test design technique where the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests.

performance testing: The process of testing to determine the performance of a software product.

product risk: A risk directly related to the test object.

quality risk: A product risk related to a quality attribute.

regression testing: Testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a result of the changes made. It is performed when the software or its environment is changed.

test approach: The implementation of the test strategy for a specific project. It typically includes the decisions made that follow based on the (test) project's goal and the risk assessment carried out, starting points regarding the test process, the test design techniques to be applied, exit criteria and test types to be performed.

test charter: A statement of test objectives, and possibly test ideas about how to test. Test charters are used in exploratory testing.

test-driven development: A way of developing software where the test cases are developed, and often automated, before the software is developed to run those test cases.

test estimation: The calculated approximation of a result related to various aspects of testing (e.g. effort spent, completion date, costs involved, number of test cases, etc.) which is usable even if input data may be incomplete, uncertain, or noisy.

test execution automation: The use of software, e.g. capture/playback tools, to control the execution of tests, the comparison of actual results to expected results, the setting up of test preconditions, and other test control and reporting functions.

test strategy: A high-level description of the test levels to be performed and the testing within those levels for an organization or programme (one or more projects).

unit test framework: A tool that provides an environment for unit or component testing in which a component can be tested in isolation or with suitable stubs and drivers. It also provides other support for the developer, such as debugging capabilities.

3.1 Agile Testing Methods

Learning Objectives

FA-3.1.1 K1 Recall the concepts of test-driven development, acceptance test-driven development, and behaviour-driven development.

FA-3.1.2 K1 Recall the concepts of the test pyramid.

FA-3.1.3 K2 Summarize the testing quadrants and their relationships with testing levels and testing types.

FA-3.1.4 K3 For a given Agile project, practice the role of a tester in a Scrum team.

There are certain testing practices that can be followed in every development project (agile or not) to help produce quality products. These include:

- writing tests in advance writing the code to better express the required behaviour;
- focusing on early defect prevention, detection, and removal; and
- ensuring that the right test types are run at the right time and as part of the right test level.

Agile practitioners aim to introduce these practices early. Testers in Agile projects play a key role in guiding the use of these testing practices throughout the lifecycle. In this subsection we focus on the testing practices that are of particular importance to Agile projects.

3.1.1 Test-Driven Development, Acceptance Test-Driven Development, and Behaviour-Driven Development

Learning Objective

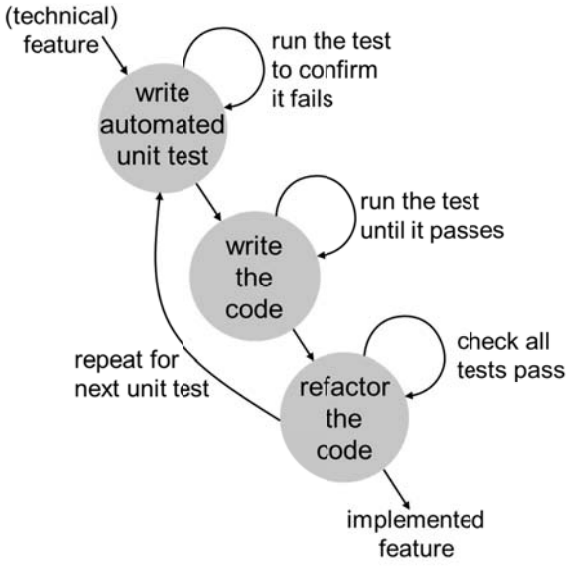
FA-3.1.1 K1 Recall the concepts of test-driven development, acceptance test-driven development, and behaviour-driven development.

Test-driven development, acceptance test-driven development, and behaviour-driven development are three complementary techniques in use among Agile teams to carry out testing across the various test levels. Each technique is an example of a fundamental principle of testing: the benefit of early testing and QA activities. They all require that the tests are defined before the code is written.

3.1.1.1 Test-Driven Development

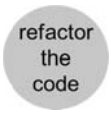
Test-driven development (TDD) is an approach to developing code that is driven by automated unit tests. , as shown in the diagram to the right.

Traditionally a programmer identifies some low-level functionality and starts to write the code that will implement it. With TDD, consideration is given to breaking down the feature to be implemented into a number of small incremental steps that can be implemented and tested separately.

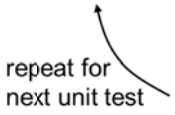


Having identified a small piece of functionality, the programmer writes an automated unit test that will verify that the intended functionality works correctly. As soon as the automated unit test is implemented it is run to confirm that it fails (since the code implementing the required functionality has not yet been written).

Next the programmer writes the code to implement the feature, writing sufficient code to pass the test but no more. The automated test is run again (along with any previously run unit tests) and hopefully confirms that the code works correctly. If not, the code is reworked and the tests are run again. This continues until the automated tests pass.



The programmer may then undertake some refactoring of the code to ensure it is of good quality (e.g. is understandable to others, contains no inappropriately repeated code and will be highly maintainable). Finally, all of the automated unit tests for the code being worked on are run to confirm that the refactoring has not introduced any unexpected changes.



The unit test for the next incremental step is automated and the TDD process continues as before. This cycle is repeated until all the unit tests applicable to the feature being implemented have been automated and the necessary code has been implemented and tested successfully.

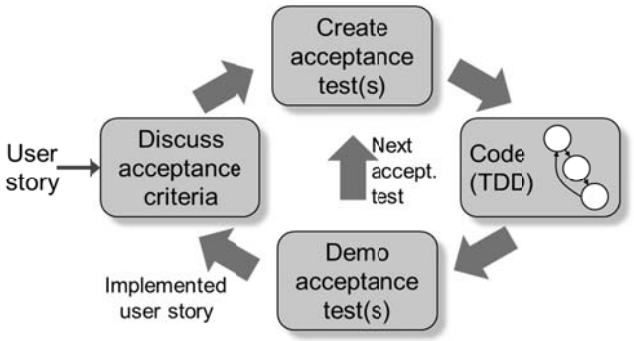
The tests written are primarily unit level and are code-focused. Tests may also be written at the integration or system levels particularly when combining previously tested units of code. Test-driven development gained its popularity through Extreme Programming [Beck02], but is also used in other Agile methodologies and sometimes in sequential lifecycles. It helps developers focus on clearly-defined expected results. Because the tests are automated they are used in continuous integration and run against each build of the code.

The TDD approach also helps to ensure that there is no untested code in the final software product. Since all code is written in response to an automated test there can be no untested code written.

The traditional code-then-test approach often falls down when the programmers are under pressure to write more code as this encourages the programmer to reduce the testing or leave it until a later time – and perhaps never getting back to do so.

3.1.1.2 Acceptance Test-Driven Development

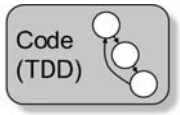
Acceptance test-driven development (ATDD) is an approach not unlike test TDD in that code is created only in response to tests (hence the 'test-driven' bit ☺). In the case of ATDD, the tests are Agile acceptance tests (i.e. at the business level) that are derived (either directly or indirectly) from the user stories.



ATDD is a collaborative approach to implementing user stories that involves the whole team. It starts when the team is ready to begin work on a given user story. This may be at the start of the iteration (such as in the iteration planning meeting or as one of the first tasks) or at some point during the

iteration. The customer or business representative describes acceptance criteria (also known as *conditions of satisfaction* or *high-level acceptance tests*) for a given user story. These may be real-world examples that state what the system should do (or not do) in specific situations. The rest of the team ask questions about the acceptance criteria to identify missing conditions and clarify understanding. This collaborative approach helps every stakeholder understand how the software has to behave.

The acceptance criteria are then converted into acceptance tests by the team or individuals within the team. If the business representative is not involved in identifying the more detailed acceptance tests, it will be necessary for them to validate acceptance tests as soon as possible after they have been written. Crucially, the tests are written in a natural language form (ideally in the business domain language). The whole team should agree with the set of acceptance tests as these become the specification for development.



The automated acceptance tests should be ready by the time the software has been written so the developers can validate their code immediately, quickly resolving any defects and confirming that the acceptance criteria for the feature have been met.

The acceptance tests are used for regression testing whenever appropriate and often within the continuous integration process. They are typically automated using specific tools to support the creation and execution of acceptance tests. These tools can connect to data and service layers of the application (for example using an API to bypass the GUI).

A more detailed description of the acceptance test-driven development process is given in Section 3.3.2 “Applying Acceptance Test-Driven Development”.

3.1.1.3 Behaviour-Driven Development

Behaviour-driven development (BDD) [Chelimsky10] is an approach much like ATDD in terms of the process being followed but differs in its focus. Like ATDD, BDD was derived from and built on TDD. BDD would be used as an alternative to ATDD.

Behaviour-driven development encourages developers to test the code based on the expected behaviour of the software rather than on implementation details. Because the tests are based on the exhibited behaviour of the software, the tests are generally easier for other team members and stakeholders to understand. They are also more robust since refactoring the code does not (is not meant to) change what the software does (its behaviour), only how it works.

The focus on behaviour is achieved by using the given / when / then template to specify acceptance criteria:

- **Given** *some initial context*,
- **When** *an event occurs*,
- **Then** *ensure some outcome*.

- **Given** I have a discount code,
- **When** I enter it on checkout,
- **Then** the purchase price will be discounted.

This can be used to describe both application behaviour (for acceptance tests) and code behaviour (for unit tests). In this way BDD helps the developer collaborate with other stakeholders, including testers, to define accurate unit tests focused on business needs.

An advantage of BDD over ATDD is that it removes the word ‘test’. Experience has shown that talking about ‘tests’ with business representatives can be hard work as some have no interest in tests. However, talk to them about behaviour of the system and they are much more interested!

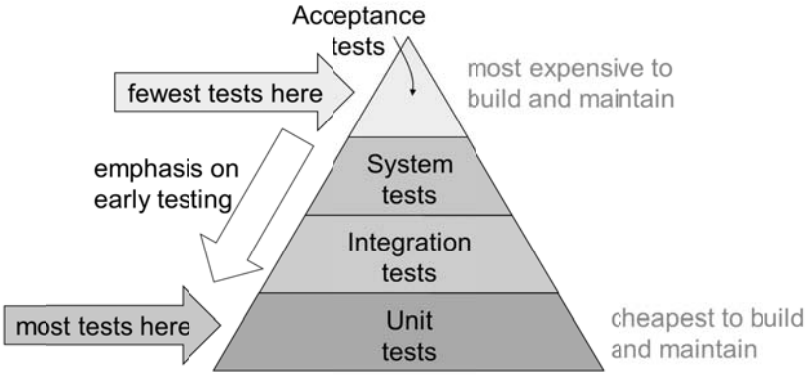
BDD has evolved with specific tool support such as RSpec and Cucumber that provide a behaviour-driven development framework. This framework generates code that can be used by developers to create automated tests.

3.1.2 The Test Pyramid

Learning Objective

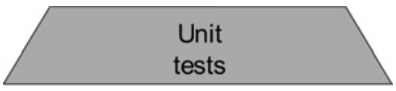
FA-3.1.2 K1 Recall the concepts of the test pyramid.

Note that [Crispin08] and [Cohn10] refer to the *test pyramid* as the *test automation pyramid*. The syllabus refers to it only as the *test pyramid* though describes it as showing levels of automated tests. Also, note that the following description uses 'Acceptance Testing' as it is defined in the Foundation Level syllabus and should not be confused with the Agile acceptance tests previously described.



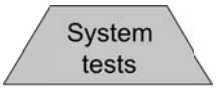
The test automation pyramid (shown above) depicts levels of testing, starting with the unit tests at the bottom of the pyramid, and progressing up through integration and system tests to acceptance tests at the top. It is the characteristics of the automated tests at each of the levels that the pyramid depicts.

The size (width) of the levels decreases as we move from the bottom to the top of the pyramid and this represents the number of automated tests at each level. The majority of automated tests should be at the unit testing level and the fewest number of automated tests (in comparison with the other levels) should be at the acceptance level. The rationale for this is that it supports the principle of early testing (i.e., eliminating defects as early as possible in the lifecycle). Tests at the unit level are run before any of the others and so will provide the fastest feedback to the developers. Also, the cost of creating, using and maintaining the automated tests is much greater for acceptance and system level tests than it is for integration and unit level tests.

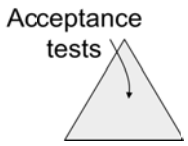


The unit tests are typically the most robust. They each focus on a specific piece of the product code and consequently when a test fails it is relatively easy to isolate the defect. The unit tests are typically automated using one of the xUnit unit test frameworks. Once the developers have mastered test-driven development the automated unit tests are generally the quickest and least expensive to write, particularly when they are tests that the customers don't have to read. As they provide the quickest feedback too, they have the greatest return on investment of any type of test [Crispin08].

The integration tests are much like the unit tests in that they are typically robust. They may also be implemented using a unit test framework and/or API tools. The API tools allow the tests to interface to software under test via the API, thereby bypassing the GUI. It is this that contributes much to the robustness of the tests. Many, if not all, of the unit and integration tests will be run with every build to provide quick and early feedback.



The system tests are much less robust than the unit and integration tests. This is because they are exercises wider capabilities of the software and are therefore more vulnerable to software changes. Also the automated system tests typically require more effort to implement and take much longer to run. They are automated using GUI-based tools.



Finally, at the top of the pyramid there are the Acceptance level tests (not to be confused with the Agile acceptance tests). This level should comprise the smallest number of automated tests. This is because they are likely to be the most expensive of the automated tests to write and maintain. Much like the system level tests, these tests interact with the software under test via the GUI, and as this is prone to change the tests are vulnerable to even the most minor of GUI changes making frequent and possibly time consuming updates necessary. They may take hours to run so cannot be run every build, and sometimes not even every night. The GUI tests have the lowest ROI.

Note that the test automation pyramid shown and described here applies to the traditional development lifecycle models as described in [ISTQB_FL_SYL] Section 2.2. The concept of the pyramid model also applies to the Agile software development model and in this context it is typically shown with three different layers of automated tests: unit tests, acceptance tests (acceptance tests in the Agile sense) and GUI tests. Note that this three level version is not covered by the Agile syllabus.

3.1.3 Testing Quadrants, Test Levels, and Testing Types

Learning Objective

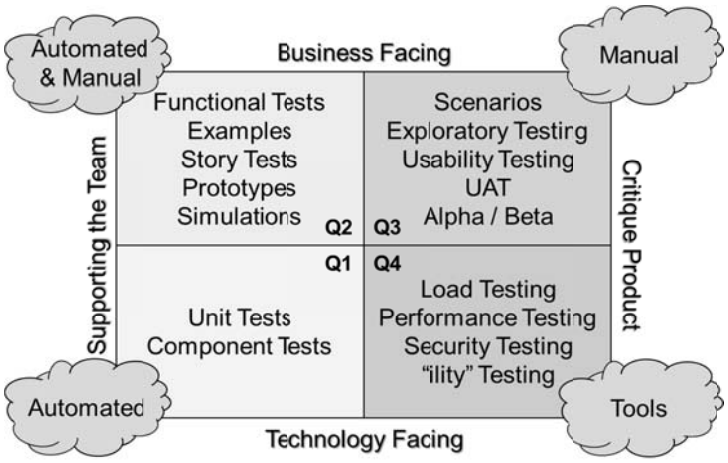
FA-3.1.3 K2 Summarize the testing quadrants and their relationships with testing levels and testing types.

The Agile testing quadrants model was defined by Brian Marick [Crispin08]. It aligns the test levels with the appropriate test types in the Agile methodology and can be used to help ensure that all important test types and test levels are considered. This model also provides a way to differentiate and describe the types of tests to all stakeholders.

The testing quadrant model puts the various types of testing used within Agile into one of four groups, represented by the four quadrants in the model. Considering the pairs of quadrants top and bottom, left and right, the quadrants in each pair share some characteristics as follows:

- along the top are business (or customer) facing tests;
- along the bottom are technology (or developer) facing tests;
- on the left-hand side are tests that support the work done by the Agile team and confirm software behaviour;
- on the right-hand side are tests that critique the product.

Tests can be fully manual, fully automated, a combination of manual and automated, or manual but supported by tools. The four quadrants are numbered arbitrarily (there is no significance in the order of the numbering).



Brian Marick's Agile Testing Quadrants

Quadrant Q1 contains unit and component level, technology facing tests that support the developers and verify the functionality of the code. These tests should be automated and

included in the continuous integration process being run on every build. They are typically automated using one of the xUnit unit test framework tools, so they are written in the same programming language as the product software. They are not likely to be understood by the business representative but the tests are not intended for them. They help measure the internal quality of the code and this is a concern of the developers.

Quadrant Q2 contains system level, business facing tests that support the team at a higher level than the Q1 tests by confirming product behaviour. These include functional tests, examples, story tests, user experience prototypes, simulations and many of the tests written using ATDD and BDD. These tests check the acceptance criteria, and can be manual or automated. Business representatives use (and write) them to define external product quality. They are often created during the user story development as an executable part of the specification and thus help to improve the quality of the stories. They are useful when creating automated regression test suites.

Quadrant Q3 contains system and user acceptance level tests. These are business facing tests that critique the product, using realistic scenarios and data. This quadrant contains exploratory testing, scenarios, process flows, usability testing, user acceptance testing, alpha testing, and beta testing. These tests are often manual and are user-oriented. For example, user acceptance testing may be used, and this will involve end-users trying out the new features. Usability testing is another example of a Q3 testing type involving end users performing tests manually. Testers on the team are likely to undertake exploratory testing to find defects that the automated tests may have missed.

Quadrant Q4 contains system and operational acceptance level tests. These are technology facing tests that critique the product. This quadrant contains performance, load, stress, and scalability tests, security tests, maintainability, memory management, compatibility and interoperability, data migration, infrastructure, and recovery testing. Some of these quality characteristics can be as important, if not more important, than some of the functionality of the system. These tests are often supported by tools but also involve much manual intervention.

During any given iteration, tests from any or all quadrants may be required. The testing quadrants apply only to dynamic testing and not static testing.

3.1.4 The Role of a Tester

Learning Objective

FA-3.1.4 K3 For a given Agile project, practice the role of a tester in a Scrum team.

Throughout these notes, general reference has been made to Agile methods and techniques, and the role of a tester within various Agile lifecycles. This subsection looks specifically at the role of a tester in a project following a Scrum lifecycle.

3.1.4.1 Teamwork

Teamwork is a fundamental principle in Agile development. Agile emphasises the whole-team approach consisting of developers, testers, and business representatives working together. The following are organisational and behavioural good practices in Scrum teams:

- Cross-functional

  Each team member brings a different set of skills to the team. The team works together on test strategy, test planning, test specification, test execution, test evaluation, and test results reporting. Testers should share their testing skills and knowledge with the other team members, and acquire new skills and knowledge from them.

- Self-organising

  It takes time for a new Agile team to learn how to work together. The team will make some mistakes along the way but it should learn from them and grow stronger.

The team may consist only of developers, but, as noted in Section 2.1.5, ideally there would be one or more testers.

- Co-located

  Agile teams should be co-located so the testers sit together with the programmers and the product owner. Having easy access to everyone on the team encourages communication. Constant visibility of a single set of progress charts keeps the team 'on the same page'. These are benefits of co-location and they support core Agile values (individuals and interactions over processes and tools) and principles (work together daily).

- Collaborative

  Testers collaborate with their team members, other teams, the stakeholders, the product owner, and the Scrum Master.

- Empowered

  Teams that are empowered to identify and solve their own problems tend to make better progress [Crispin08]. Technical and prioritising decisions regarding design and testing are made by the team as a whole (developers, testers, and Scrum Master), in collaboration with the product owner and other teams if needed. (Note that the product owner is not necessarily involved in technical decisions.)

- Committed

  The tester is committed to question and evaluate the product's behaviour and characteristics, with respect to the expectations and needs of the customers and users. This is a key role that testers bring to the Agile team: being an advocate for the customers.

- Transparent

  Programming and testing progress is visible on the Agile task board (see Section 2.2.1).

- Credible

  The tester must ensure the credibility of the strategy for testing, its implementation, and execution; otherwise the stakeholders will not trust the test results. This is often done by providing information to the stakeholders about the testing process.

- Open to feedback

  Feedback is an important aspect of being successful in any project, especially in Agile projects. Retrospectives allow teams to learn from successes and from failures.

- Resilient

  Testing must be able to respond to change, like all other activities in Agile projects.

These best practices maximise the likelihood of successful testing in Scrum projects.

3.1.4.2 Sprint Zero

Sprint zero is the first sprint of the project where many preparation activities are taking place (Section 1.2.5). The tester collaborates with the team on the following activities during this sprint:

- Identify the scope of the project (i.e., the product backlog).
- Create an initial system architecture and high-level prototypes.
- Plan, acquire, and install needed tools (e.g., for test management, defect management, test automation, and continuous integration).
- Create an initial test strategy for all test levels, addressing (among other topics): test scope, technical risks, test types (see Section 3.1.3), and coverage goals.
- Perform an initial quality risk analysis (see Section 3.2.1).

- Define test metrics to measure the test process, the progress of testing in the project, and product quality.
- Specify the definition of “done”.
- Create the task board (see Section 2.2.1).
- Define when to continue testing or stop testing and deliver the system to the customer.

Sprint zero sets the direction for what testing needs to achieve and how testing needs to achieve it throughout the sprints.

3.1.4.3 Integration

As with any Agile project, the objective of Scrum projects is to deliver customer value on a continuous basis (preferably in every sprint). To help achieve this, the integration strategy should consider both design and testing. Testers therefore must ensure that appropriate tests are automated and scheduled to be run at the most appropriate stages of integration. It will also be necessary to identify all dependencies between functions and the quality characteristics being tested.

3.1.4.4 Test Planning

Since testing is fully integrated into the Agile team, test planning should start during the sprint zero (release planning) and be updated during each sprint. Test planning for the release and each sprint should address issues such as:

- the scope of testing;
- identifying team members responsible for the various test activities;
- test environment and test data requirements;
- timing, sequencing, dependencies and prerequisites for the testing;
- project and quality risks to be addressed (assessment and mitigation).

These are further discussed in Section 1.2.5.

Sprint planning results in a set of tasks to put on the task board, where each task should have a length of one or two days of work. In addition, any testing issues should be tracked to maintain a steady flow of testing.

3.1.4.5 Agile Testing Practices

Many practices may be useful for testers in a scrum team, some of which include:

- Pair testing

  Two team members (e.g., a tester and a developer, two testers, or a tester and a product owner) sit together at one workstation to perform a testing task.

- Incremental test design

  Test cases and charters (for exploratory testing) are gradually built from user stories and other test basis, starting with simple tests and moving toward more complex ones.

- Mind mapping

  Mind mapping is a useful tool when testing [Crispin08]. For example, testers can use mind mapping to identify which test sessions to perform, to show test strategies, and to describe test data.

These practices are in addition to other practices discussed in this syllabus and in Chapter 4 of the Foundation Level syllabus [ISTQB_FL_SYL].

3.2 Assessing Quality Risks and Estimating Test Effort

Learning Objectives

FA-3.2.1 K3 Assess quality risks within an Agile project.

FA-3.2.2 K3 Estimate testing effort based on iteration content and quality risks.

A typical objective of testing in many projects, Agile or traditional, is to remove as many defects as possible from the product prior to release. The purpose of this is, of course, to ensure as few defects as possible are found by the users after release. However, it is often also a requirement that this reduction in defects be achieved within a constrained time and cost.

This gives rise to one of the many challenges in testing: the proper selection, allocation, and prioritization of tests. It is necessary to determine the appropriate amount of effort to allocate to testing and then sequence the resulting tests in a way that optimizes the effectiveness and efficiency of the testing work to be done.

One possible approach that can help to solve this problem is to use risk, that is, risk-based testing. The focus is then on reducing the risk of product quality problems to an acceptable level prior to release.

Testers in Agile projects can use the same types of techniques used in traditional projects to manage risks. Risk management includes identifying risks, assessing the associated level of risk, estimating the effort required to reduce the risks sufficiently, and mitigating those risks through test design, implementation, and execution. However, given the short iterations and rate of change in Agile projects, some adaptations of those techniques are required.

3.2.1 Assessing Quality Risks in Agile Projects

Learning Objective

FA-3.2.1 K3 Assess quality risks within an Agile project.

Risk identification, analysis, and mitigation strategies can be used by the testers in Agile teams to help determine an acceptable number of test cases to execute, although many interacting constraints and variables may require compromise.

What is risk?

Risk is the possibility of a negative or undesirable outcome or event. A risk exists whenever some problem may occur which would decrease customer, user, participant, or stakeholder perceptions of product quality or project success. Risks are not actual problems – they are problems that may occur in the future, so there is a chance that it will not happen and there is the inverse chance that it will happen. To indicate how much risk there is we typically determine the 'risk level' (also known as the 'risk exposure'). This is a combination of the chance or likelihood of the problem occurring and the impact that the problem will have if it does occur. The more likely it is that a problem will occur and/or the greater the (negative) impact the problem will have, the greater the risk level.

Risks that become problems (i.e. the problem occurs) are said to have 'matured'. So a risk that matures is no longer a risk, it's a problem!

Where the primary effect of the potential problem (risk) is on product quality, potential problems are referred to as '**quality risks**', '**product risks**', or '**product quality risks**'. Where the primary effect of the potential problem is on project success, potential problems are referred to as '**project risks**' or '**planning risks**'. [Black07] [vanVeenendaal12]

Identifying and assessing risks

In Agile projects, risk identification and analysis takes place within two specific activities:

- Release planning

  Business representatives who know the features in the release provide a high-level overview of the risks, and the whole team, including the tester(s), may assist in the risk identification and assessment.

- Iteration planning

  During the iteration planning meeting the whole team works together to identify and assess quality risks. The outstanding risks identified during release planning and earlier iteration planning meetings will also be reviewed.

Examples of quality risks include:

- Calculations presented in reports may be incorrect.

  This is a functional risk related to accuracy.

- Response times may be too slow.

  This is a non-functional risk related to efficiency and response time.

- Users may have difficulty in understanding screens and fields.

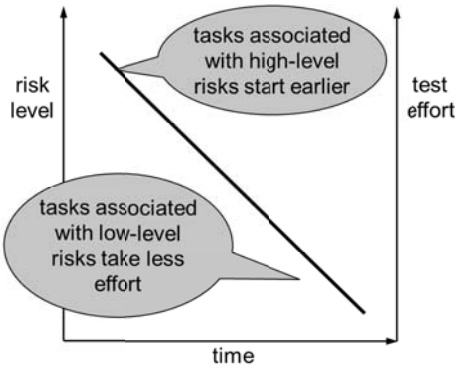
  This is a non-functional risk related to usability and understandability.

Examples of project risks include:

- Delivery of the hardware for the test environment may be late.
- A member of the team may resign.
- Construction work outside the office may distract the team.

Project risks are traditionally management issues but an Agile team should be sufficiently empowered to address them, at least initially.

As mentioned earlier, an iteration starts with iteration planning, which culminates in estimated tasks on a task board. These tasks can be prioritized in part based on the level of quality risk associated with them. Tasks associated with higher risks should start earlier and involve more testing effort. Tasks associated with lower risks should start later and involve less testing effort.



Example risk analysis process

An example of how the quality risk analysis process in an Agile project may be carried out during iteration planning is outlined in the steps below (this list has been taken from the syllabus).

1. Gather the Agile team members together, including the tester(s).
2. List all the backlog items for the current iteration (e.g., on a task board).
3. Identify the quality risks associated with each item, considering all relevant quality characteristics.
4. Assess each identified risk, which includes two activities: categorizing the risk (e.g. functional, usability) and assigning a level of risk based on the impact and the likelihood.
5. Determine the extent of testing proportional to the level of risk.
6. Select the appropriate test technique(s) to mitigate each risk, based on the category of risk and the level of risk.

The tester then designs, implements, and executes tests to mitigate the risks. This includes the totality of features, behaviours, quality characteristics, and attributes that affect customer, user, and stakeholder satisfaction.

Risk re-evaluation

Throughout the project, the team should remain aware of additional information that may change the set of risks and/or the level of risk associated with known quality risks. Periodic adjustment of the quality risk analysis, which results in adjustments to the tests, should occur. Adjustments include identifying new risks, re-assessing the level of existing risks, and evaluating the effectiveness of risk mitigation activities.

Static testing

Quality risks can also be mitigated before test execution starts. For example, if problems with the user stories are found during risk identification, the project team can undertake a more thorough review of user stories as a mitigating strategy.

3.2.2 Estimating Testing Effort Based on Content and Risk

Learning Objective

FA-3.2.2 K3 Estimate testing effort based on iteration content and quality risks.

During release planning, the Agile team estimates the effort required to complete the release. During iteration planning the team estimates the effort required to implement (including the testing of) individual user stories. This leads to a determination of how many of the user stories can be implemented within the fixed time of the iteration.

Story points

Estimates are often given in numbers called ‘story points’. These give an indication of the work needed to implement a story. It may be a relative indication of the size or complexity of a story (relative to other stories). So a user story estimated to be 10 story points is expected to take twice as long as a story estimated to be 5 story points.

Story points are equated to a meaningful measure such as ‘effort days’, effort weeks, or some other units the team decides to use. (The team will normally decide on the units for estimation during release planning.)

A distinction can be made between elapsed time and ideal time. If we equate story points to a measure of elapsed time, the estimate must take into consideration all the possible interruptions team members will experience throughout the working day: answer emails, taking phone calls, company meetings, etc. Using ideal time means that for the purpose of the estimate we ignore the interruptions and assume that each day will be as productive and uninterrupted as possible. It is easier to estimate using ideal time, though this will have to be converted to elapsed time eventually.

Note that one team’s story points are unlikely to be the same as those of another team. So we should not compare measures of story points across different teams.

Planning poker

A common estimation technique used in Agile projects is called planning poker, a consensus-based technique. It involves the product owner and a selection of the team, both programmers and testers. If it is a small team then probably the whole team should be involved.

Each estimator has a deck of cards with values similar to the Fibonacci sequence (i.e., 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …) or any other progression of choice (e.g., shirt sizes ranging from extra-small to extra-extra-large). The Fibonacci sequence is recommended because the numbers in the sequence reflect that uncertainty grows proportionally with the size of the story. (A high estimate usually means that the story is not well understood or should be broken down into multiple smaller stories.)

The product owner or customer reads a user story to the estimators. The estimators discuss the feature, and ask questions of the product owner as needed. Aspects such as development and testing effort, complexity of the story, and scope of testing all play a role in

the estimation. Therefore, it is advisable to include the risk level of a backlog item, in addition to the priority specified by the product owner, before the planning poker session is initiated.

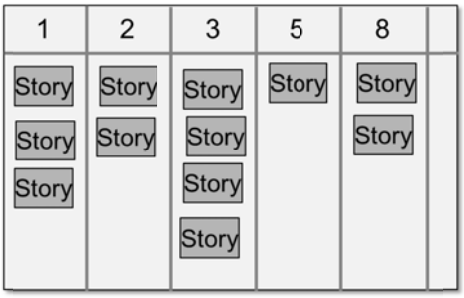
When the feature has been fully discussed, each estimator privately selects one card to represent his or her estimate. All cards are then revealed at the same time. If all estimators selected the same value, that becomes the estimate. If not, the estimators discuss the differences in estimates; in particular, those who gave the highest or lowest estimates should explain their thinking. After this second round of discussion the poker round is repeated. The cycle is repeated until agreement is reached either by consensus or by applying rules (e.g., use the median, use the highest score) to limit the number of poker rounds.

To keep the estimation process as objective as possible, it is important that the product owner does not contribute any estimates, or indeed, argue against the estimates made.

The discussions between the team members and the product owner ensure a reliable estimate of the effort needed to complete product backlog items requested by the product owner. They also help improve collective knowledge of what has to be done [Cohn04].

Triangulation

Mike Cohn [Cohn04] describes triangulation as a technique for checking consistency between the estimates for stories. The idea is to compare stories according to their estimated story points. “Is this story of 8 points similar to the others of 8 points, twice those of 4 points, and half those of 16 points?”. This can be best managed by pinning each story to a board organised into columns for each points estimate as shown below. Every time a new story is estimated the team should compare it with those already in place on the board.



When stories cannot be estimated

Mike Cohn [Cohn04] gives three common reasons why a team may not be able to estimate a user story. These are:

- a lack of domain knowledge

  If any team members do not understand the user story then they are unlikely to give a realistic estimate. In this situation the story should be discussed with the person who wrote the story to gain a better understanding of it.

- a lack of technical knowledge

  If the team has no knowledge of the technology involved they may need to undertake some investigation work (sufficient to be able to make an estimate of effort). This typically means the user story is turned into two stories, the first is a short investigation into the technology, the second the original user story. The second of these stories is estimated after the first has been completed.

- the user story is too big

  Epics have to be broken down into smaller stories that can be better understood and completed within a reasonable amount of time.

3.3 Techniques in Agile Projects

Learning Objectives

FA-3.3.1	K3	Interpret relevant information to support testing activities.
FA-3.3.2	K2	Explain to business stakeholders how to define testable acceptance criteria.
FA-3.3.3	K3	Given a user story, write acceptance test-driven development test cases.
FA-3.3.4	K3	For both functional and non-functional behaviour, write test cases using black box test design techniques based on given user stories.
FA-3.3.5	K3	Perform exploratory testing to support the testing of an Agile project.

Many of the test techniques and testing levels that apply to traditional projects can also be applied to Agile projects. However, for Agile projects, there are some specific considerations and variances in test techniques, terminologies, and documentation that should be considered.

3.3.1 Acceptance Criteria, Adequate Coverage, and Other Information for Testing

Learning Objectives

FA-3.3.1	K3	Interpret relevant information to support testing activities.
FA-3.3.2	K2	Explain to business stakeholders how to define testable acceptance criteria.

Requirements

As described in Section 1, Agile projects outline initial requirements as user stories in a prioritized backlog at the start of the project. User stories are short and usually follow a predefined format (see Section 1.2.2).

Non-functional requirements, such as usability and performance, are also important and can be specified as unique user stories or connected to other functional user stories. Non-functional requirements (also known as quality characteristics or quality attributes) may also follow a predefined format or a standard (such as the international standard [ISO25000]) or an industry specific standard. For example, the ISO standard defines a number of specific quality characteristics and sub-characteristics such as:

Characteristic	Sub-Characteristics
Usability	Understandability Learnability Operability Attractiveness
Efficiency	Performance Resource Use

The user stories serve as an important test basis. However, they are not the only source of information for testing. Other possible test bases include:

- Experience from previous projects

  Both good and bad experiences of developing and testing software can inspire ideas for testing.

- Existing functions, features, and quality characteristics of the system

  How one part of a system behaves will influence our understanding of behaviour of new software.

- Code, architecture, and design

  Specific code-based testing techniques or test approaches applicable to architectural and design solutions.

- User profiles (context, system configurations, and user behaviour)

  Having a better understanding of how a user is likely to behave can help identify tests that help ensure the software better meets the user needs.

- Quality risks (see Section 3.2.1)

  Adequate mitigation of quality risks can only be achieved by appropriate test approaches and techniques.

- Information on defects from existing and previous projects

  Defects already found hint at possible future defects so considering tests for previous defects may be effective.

- A categorization of defects in a defect taxonomy

  Defects typical of the type of software or system being developed can be a good source of test ideas.

- Applicable standards (e.g., [DO-178B] for avionics software)

  An independent but relevant source of test ideas, particularly where compliance is an issue.

During each iteration developers create code that implements the functions and features as described by the user stories together with acceptance criteria, what is learnt from or thought of during conversations, and notes of past conversations.

This code is verified and validated via unit testing and acceptance testing.

Acceptance criteria

To be testable, acceptance criteria should address the following topics where relevant [Wiegers13]:

- **Functional behaviour**: The externally observable behaviour with user actions as input operating under certain configurations.
- **Quality characteristics**: How the system performs the specified behaviour. The characteristics may also be referred to as quality attributes or non-functional requirements. Common quality characteristics are performance, reliability, usability, security, etc.
- **Scenarios (use cases)**: A sequence of actions between an external actor (often a user) and the system, in order to accomplish a specific goal or business result.
- **Business rules**: Activities that can only be performed in the system under certain conditions defined by outside procedures and constraints (e.g., the procedures used by an insurance company to handle insurance claims).
- **External interfaces**: Descriptions of the connections between the system to be developed and the outside world. External interfaces can be divided into different types (user interface, interface to other systems, etc.).
- **Constraints**: Any design and implementation constraints that will restrict the options for the developer. Devices with embedded software must often respect physical constraints such as size, weight, and interface connections.
- **Data definitions**: The customer may describe the format, data type, allowed values, and default values for a data item in the composition of a complex business data structure (e.g., the ZIP code in a U.S. mail address).

Other information

In addition to the user stories and their associated acceptance criteria, other information is relevant for the tester, including:

- How the system is supposed to work and be used
- The system interfaces that can be used/accessed to test the system
- Whether current tool support is sufficient
- Whether the tester has enough knowledge and skill to perform the necessary tests

Testers will often discover a need for additional information (e.g., code coverage) throughout the iterations and should work collaboratively with the rest of the Agile team members to obtain that information.

Definitions of 'done'

'Done' defines when a task or activity has been completed. The definition of 'done' is a critical aspect of Agile projects. It is typically defined by the team as part of the release planning activity.

Typically there will be different definitions of 'done' defined for different activities as discussed in the following sub-subsections.

3.3.1.1 Test Levels

Each test level has its own definition of done. The following list gives examples that may be relevant for the different test levels.

- Unit testing
  - 100% decision coverage where possible, with careful reviews of any infeasible paths.
  - Static analysis performed on all code.
  - No unresolved major defects (ranked based on priority and severity).
  - No known unacceptable technical debt remaining in the design and the code [Jones11].
  - All code, unit tests, and unit test results reviewed.
  - All unit tests automated.
  - Important characteristics are within agreed limits (e.g., performance).
- Integration testing
  - All functional requirements tested, including both positive and negative tests, with the number of tests based on size, complexity, and risks.
  - All interfaces between units tested.
  - All quality risks covered according to the agreed extent of testing.
  - No unresolved major defects (prioritized according to risk and importance).
  - All defects found are reported.
  - All regression tests automated, where possible, with all automated tests stored in a common repository.
- System testing
  - End-to-end tests of user stories, features, and functions.
  - All user personas covered.
  - The most important quality characteristics of the system covered (e.g., performance, robustness, reliability).
  - Testing done in a production-like environment(s), including all hardware and software for all supported configurations, to the extent possible.
  - All quality risks covered according to the agreed extent of testing.
  - All regression tests automated, where possible, with all automated tests stored in a common repository.
  - All defects found are reported and possibly fixed.
  - No unresolved major defects (prioritized according to risk and importance).

3.3.1.2 User Story

The definition of done for user stories may be determined by the following criteria:

- The user stories selected for the iteration are complete, understood by the team, and have detailed, testable acceptance criteria.
- All the elements of the user story are specified and reviewed, and user story acceptance tests have been completed.
- Tasks necessary to implement and test the selected user stories have been identified and estimated by the team.

3.3.1.3 Feature

The definition of done for features, which may span multiple user stories or epics, may include:

- All constituent user stories, with acceptance criteria, are defined and approved by the customer.
- The design is complete, with no known technical debt.
- The code is complete, with no known technical debt or unfinished refactoring.
- Unit tests have been performed and have achieved the defined level of coverage.
- Integration tests and system tests for the feature have been performed according to the defined coverage criteria.
- No major defects remain to be corrected.
- Feature documentation is complete, which may include release notes, user manuals, and on-line help functions.

3.3.1.4 Iteration

The definition of done for the iteration may include the following:

- All features for the iteration ready and individually tested according to the feature level criteria.
- Any non-critical defects that cannot be fixed within the constraints of the iteration added to the product backlog and prioritized.
- Integration of all features for the iteration completed and tested.
- Documentation written, reviewed, and approved.

At this point, the software is potentially releasable because the iteration has been successfully completed, but not all iterations result in a release.

3.3.1.5 Release

The definition of done for a release, which may span multiple iterations, may include the following areas:

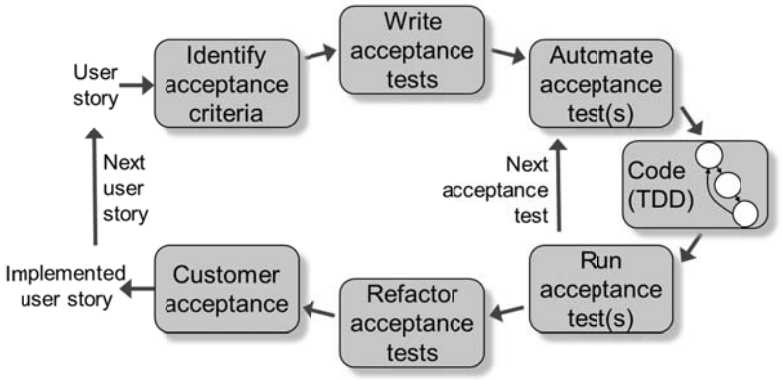
- **Coverage**: All relevant test basis elements for all contents of the release have been covered by testing. The adequacy of the coverage is determined by what is new or changed, its complexity and size, and the associated risks of failure.
- **Quality**: The defect intensity (e.g., how many defects are found per day or per transaction), the defect density (e.g., the number of defects found compared to the number of user stories, effort, and/or quality attributes), the estimated number of remaining defects are within acceptable limits, the consequences of unresolved and remaining defects (e.g., the severity and priority) are understood and acceptable, the residual level of risk associated with each identified quality risk is understood and acceptable.
- **Time**: If the pre-determined delivery date has been reached, the business considerations associated with releasing or not releasing need to be considered.
- **Cost**: The estimated lifecycle cost should be used to calculate the return on investment for the delivered system (i.e., the calculated development and maintenance cost should be considerably lower than the expected total sales of the product). The main part of the lifecycle cost often comes from maintenance after the product has been released, due to the number of defects escaping to production.

3.3.2 Applying Acceptance Test-Driven Development

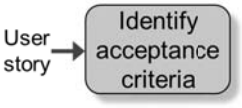
Learning Objective

FA-3.3.3 K3 Given a user story, write acceptance test-driven development test cases.

Acceptance test-driven development is a test-first approach, meaning that tests are created prior to implementing the user story (or that part of the user story to which the acceptance test applies). When coding, the aim is to write sufficient code to pass the tests but no more. This helps to ensure that all product code is tested and that there is no unnecessary code introduced (i.e. code that does not provide business value).



Identify acceptance criteria



ATDD starts when the team chooses a user story to implement (usually the highest priority user story remaining in the iteration backlog). The team meet to analyse and discuss the user story with the business representative.

Business rules have to be defined by the business representatives because they best understand what is required. To ensure that the initial information is good enough for development and testing to implement those rules correctly, the business representatives and the rest of the team have to work together to refine the definition of them.

The discussion builds up a shared understanding of *what the software should do* (i.e. the requirement) and should avoid attempts to specify *how* it is to work (i.e. giving a solution). Specifying solutions at this stage unnecessarily constrains the developers and may prevent them from coming up with innovative solutions.

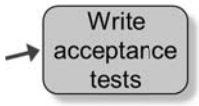
The business representative expands on the user story by giving some acceptance criteria – statements about what the product is expected to do from a business perspective in specific situations. Ideally the business representative will have thought about and prepared an initial set of acceptance criteria beforehand.

The rest of the team ask questions about the user story and the acceptance criteria to uncover gaps and contradictions, and to identify additional examples or acceptance criteria that the business representative may not have thought about. Any incompleteness, ambiguities, or other defects in the user story are fixed during this discussion. The goal is to identify the acceptance criteria that will help the developers better understand the business representative's expectations and serve as the means of determining that the implementation of the user story is correct and complete.

Note that the acceptance criteria we are discussing here may have different names and may be written in different forms. A common alternative name is *conditions of satisfaction* or *high-level acceptance tests*. They may also be written in the form of *examples*. Goyko Adzic advocates the use of real-world examples rather than abstract requirements because discussing realistic situations helps flush out unstated assumptions [Adzic09]. The use of examples to elaborate requirements in general gives rise to Example-Driven Development

(EDD) and Specification by Example (SBE). These are approaches to ATDD that mandate the use of examples as a means of elaborating the user story.

Write acceptance tests



The next step is to create the acceptance tests. This means turning each of the acceptance criteria into one or more executable tests, specifying pre-conditions and inputs, expected outcomes and post-conditions. The acceptance tests are expressed in a way that every stakeholder (business representative, developer and tester) is able to understand. The tests are typically described by natural language statements involving any necessary preconditions, inputs, outputs and post-conditions. They may follow a template such as

Given <some precondition(s)>,
When <this happens>,
Then <this occurs>

e.g. **Given** low stock levels,
When the customer orders more than is in stock,
Then a partial order is sent.

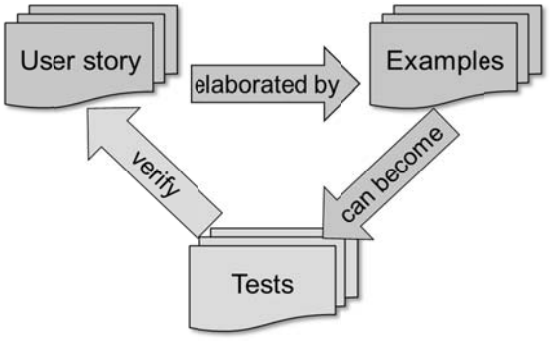
This can be done by the team together or individually by any member of the team (developer, tester, and business representative). In any case, an independent person such as the business representative validates the tests as soon as possible after they have been written and before the coding of the implementation starts. (Only acceptance tests that have been agreed by the business representative should be implemented. This helps to ensure that the team's work is continually focused on the objectives of the business representative.)

Typically, the first tests are the positive tests, confirming the correct behaviour without exception or error conditions. These invoke the sequences of actions taken when everything goes as expected. After the positive path tests are done, the team should write negative path tests and cover any important non-functional attributes as well (e.g., performance, usability).

The acceptance tests should cover all the important characteristics of the user story without adding to the story. This means that an acceptance test should not exist that describes an aspect of the user story not documented in the story itself. In addition, no two acceptance tests should describe the same characteristic of the user story.

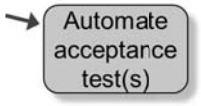
The acceptance tests are based on acceptance criteria (high-level acceptance tests, conditions of satisfaction, or examples) that elaborate the user story by describing specific characteristics of it. The examples and tests together help the team implement the user story correctly.

Note that the syllabus refers to the tests as examples and states 'Since examples and tests are the same, these terms are often used interchangeably'. This may be a bit misleading. Although the terms may be used interchangeably, they do have distinct meanings. The diagram below represents the relationship between a user story, examples that elaborate that user story and that can be used as the basis for tests, which in turn are used to verify the implementation of the user story. The 'Examples' shown in this diagram could equally be 'acceptance criteria' or 'high-level acceptance tests'.



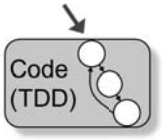
Source: Adapted from [Adzic09] referencing an original created at the 2nd Agile Alliance Functional Testing Tools Workshop, Toronto, 2008

Automate acceptance test(s)



Many of the acceptance tests written in the previous step will now be automated. This may happen one, or a few, at a time, allowing the implementation (code) step to be carried out to produce the code that will pass the first acceptance test(s) before the next acceptance test(s) are automated.

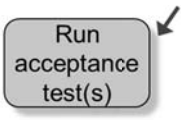
Code



Any approach to coding can be undertaken here (as far as ATDD is concerned) providing the only code that is written is that which is sufficient to pass the acceptance tests. Most likely, however, a TDD approach will be used as this has similar principles to those of ATDD; they are complementary approaches.

Typically each acceptance test will be broken down into a number of small development steps, which will be implemented and unit tested before moving on to the next one.

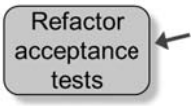
Run acceptance test(s)



As soon as the code has been written and it passes all of its unit tests, the acceptance test(s) are run to confirm (or otherwise) that the implemented code behaves as expected by the business representative. If not, the failing tests are investigated, the defects corrected and the acceptance tests are re-run. This cycle repeated until all the acceptance tests pass.

If only one or a subset of the acceptance tests have been implemented, the work continues with the 'automate acceptance test(s)' step.

Refactor acceptance tests



Once all the acceptance tests have been implemented and passed, consideration can be given to refactoring the automated acceptance tests. This is to prevent technical debt from accruing. For example there may be some duplicated code with the automated test implementation that can be extracted and made into a reusable code module.

Customer acceptance

The completed user story is now demonstrated to the business representative by running all the acceptance tests and any additional functional regression tests appropriate to the software being tested. Additional manual exploratory tests may be performed at this stage.

3.3.3 Functional and Non-Functional Black Box Test Design

Learning Objective		
FA-3.3.4	K3	For both functional and non-functional behaviour, write test cases using black box test design techniques based on given user stories.

In Agile testing, many tests are created by testers concurrently with the developers' programming activities. While the programmers are focused on creating unit tests and the code that will pass those tests, testers are likely to be focused on creating acceptance tests. Both programmers and testers will be guided by the user stories and acceptance criteria as a basis for their tests. User stories may be used to document non-functional requirements in addition to functional requirements.

Additional tests, such as those concurrently designed and executed during exploratory testing and some other experience-based testing approaches, are created during test execution, as explained in Section 3.3.4 below.

When creating their tests for functional requirements and some non-functional requirements, testers and programmers can apply traditional black-box (behavioural or specification-based) test design techniques. The Foundation Level syllabus [ISTQB_FL_SYL] described the following black-box test design techniques:

- equivalence partitioning
- boundary value analysis
- decision tables
- state transition testing

As examples, the following table suggests one or more black-box test design techniques appropriate for the given user stories.

User story	Test design technique	Comments
"As an online customer I want to order multiple products together so I have only one payment to make."	equivalence partitioning	Test a typical number of items, more than the maximum and if possible, a negative number of items.
	boundary value analysis	Test minimum and maximum number of items. Also zero and one more than the maximum?
"As the Finance Director, I want the sales figures to be accurate at least 99.9% of the time so I have reliable information on which to base my decisions."	boundary value analysis	Good technique for finding rounding defects in numeric calculations.
"As a sales agent, I want the system to calculate the correct fare for journeys according to the fare rules."	decision tables	Good for identifying and checking against combinations of conditions (business rules).
"As a hotel manager, I want guests to be able to check-in at the automated kiosk without any problems."	state transition testing	Good for exploring navigation paths through a GUI.

The user story might contain performance or reliability requirements. For example, "a given execution cannot exceed a time limit" or "a number of operations may fail less than a certain number of times".

For more information about the use of black box test design techniques, see the Foundation Level syllabus [ISTQB_FL_SYL] and the Advanced Level Test Analyst syllabus [ISTQB_ALTA_SYL].

Note that both programmers and testers may use structure-based (white-box) test design techniques in addition to the black-box techniques discussed above. These provide an objective and systematic way of adding additional tests that add value and complement the existing black-box techniques. Their use though is outside the scope of the Agile Tester syllabus. For more information see the Foundation Level syllabus [ISTQB_FL_SYL] and the Advanced Level Technical Test Analyst syllabus [ISTQB_ALTTA_SYL].

3.3.4 Exploratory Testing and Agile Testing

Learning Objective

FA-3.3.5 K3 Perform exploratory testing to support the testing of an Agile project.

Exploratory testing has been defined as:

"An approach to software testing that emphasizes the personal freedom and responsibility of each tester to continually optimize the value of his work by treating learning, test design and test execution as mutually supportive activities that run in parallel throughout the project."
(James Bach / Michael Bolton edit of a Cem Kaner suggestion.)

It is an approach or mind-set, not a specific testing technique. It requires good test design (using appropriate methods), careful observation (of the software behaviour), critical thinking (reacting with reasoned actions), diverse ideas (going beyond the obvious) and rich resources (tools, techniques, data and information sources).

Exploratory testing is important in Agile projects due to the limited time available for test analysis and the limited details of the user stories. In order to achieve the best results, exploratory testing should be combined with other experience-based techniques as part of a reactive testing strategy, blended with other testing strategies such as analytical risk-based testing, analytical requirements-based testing, model-based testing, and regression-averse testing. Test strategies and test strategy blending is discussed in the Foundation Level syllabus [ISTQB_FL_SYL].

In exploratory testing, tests are designed and executed at the same time, typically within time-boxed sessions. The sessions are usually limited to between 60 and 120 minutes and may be undertaken in pairs (testers and/or developers working together). During the exploratory testing sessions, the results of the most recent tests guide the next test. The same black-box and white-box techniques can be used to design the tests as when performing pre-designed testing though they would normally not be documented.

Test charter

A test charter provides an outline of the testing to be undertaken in an exploratory testing session, typically including some (but not all) of the following:

- **Purpose**: the theme of the charter including what particular objective the actor wants to achieve, i.e., test conditions and/or specific features, characteristics to explore or defects to retest and regression test.
- **Priority**: relative importance of this charter, based on the priority of the associated user story or the risk level.
- **Reference**: specifications (e.g., user story), risks, or other information sources.
- **Setup**: what needs to be in place in order to start the test execution.
- **Data**: whatever data that may be needed to carry out the charter.
- **Actor**: intended user of the system.
- **Activities**: a list of ideas of what the actor may want to do with the system (e.g., "Log on to the system as a super user") and what would be interesting to test (both positive and negative tests).
- **Oracle notes**: how to evaluate the product to determine correct results (e.g., to capture what happens on the screen and compare to what is written in the user's manual).
- **Variations**: alternative actions and evaluations to complement the ideas described under activities.

Managing exploratory testing

To manage exploratory testing, a method called session-based test management can be used. A session is an uninterrupted time-boxed period of testing that can last from 60 to 120 minutes. Each session has its own test charter (see above) and generates its own session report. The session report records important information about the session such as the area tested, bugs found, issues noted and time spent on setup, testing and non-testing activities. (See 'Tester notes' below.) After the session the tester has a short meeting with the manager (debrief) to report the results and decide what remains to be done.

Test sessions can be categorised according to their specific purpose. Three common categories are:

- Survey session

  The focus of the session is to learn how the software product works, what it does and what it doesn't do, preparing for formal testing (by identifying risks to testing) and/or quickly characterising the product's quality.

- Analysis session

  The focus of the session is to decide where certain risks are, what should be covered and how it should be covered. It may also focus on specific functionality or quality characteristics and how well they satisfy the business needs.

- Deep coverage session

  The focus of the session is very specific, such as a performance of a particular feature, or corner cases: things that occur outside normal operating conditions and/or that some people may perceive as unrealistic, scenarios, or interactions.

Test quality

The quality of the testing depends on the tester's ability to ask relevant questions about what to test. Examples include the following:

- What is most important to find out about the system?
- In what ways may the system fail?
- What happens if …?
- What should happen when …?
- Are customer needs, requirements, and expectations fulfilled?
- Is it possible to install the system (and remove if necessary) in all supported upgrade paths?

During test execution, the tester uses creativity, intuition, cognition, and skill to find possible problems with the product. The tester also needs to have good knowledge and understanding of the software under test, the business domain, how the software is to be used, and how to determine when the system fails.

Heuristics

There are many heuristics (guidelines, checklists, rules of thumb) applied to software testing. A heuristic can guide the tester to think of new test conditions [Hendrickson2]. Here are a few examples:

- Boundaries

  Try values close to and on the boundaries of a range: almost too big, almost too small, at the boundary.

- Configuration variations

  Varying the variables related to configuration (e.g. screen resolution, network speed, latency, memory, disk availability, etc.)

- CRUD (Create, Read, Update, Delete)

  Variety of actions on data.

- Interruptions

  Various forms of interruption (e.g., log off, shut down, reboot, kill process, disconnect, hibernate, timeout, cancel).

- Selection

  None, some, all.

- Sequences

  Vary the order of operations, undo/redo, reverse, invert, combine, simultaneous.

Tester notes

It is important for the tester to document the process ‘as much as necessary’. Note that the syllabus says ‘as much as possible’ but this is contradictory to the aims of exploratory testing (to do testing rather than document it). The tester must record sufficient information to enable the tester to go back and see how a problem in the system was discovered, be able to report progress objectively and follow-up on issues discovered and questions thought of during the session. In terms of session-based testing this is known as the session report.

The following list provides examples of information that may be useful to document:

- **Test coverage**: what input data have been used, how much has been covered, and how much remains to be tested.
- **Evaluation notes**: observations during testing, do the system and feature under test appear to be stable, were any defects found, what is planned as the next step according to the current observations, and any other list of ideas.
- **Risk/strategy list**: which risks have been covered and which ones remain among the most important ones, will the initial strategy be followed, does it need any changes.
- **Issues, questions, and anomalies**: any unexpected behaviour, any questions regarding the efficiency of the approach, any concerns about the ideas/test attempts, test environment, test data, misunderstanding of the function, test script or the system under test.
- **Actual behaviour**: recording of actual behaviour of the system that needs to be saved (e.g., video, screen captures, output data files).

The information logged can be captured and/or summarized into some form of status management tool (e.g., test management tool, task management tool, or the task board), in a way that makes it easy for stakeholders to understand the current status for all testing that was performed.

3.4 Tools in Agile Projects

FA-3.4.1	K1	Recall different tools available to testers according to their purpose and to activities in Agile projects.

Tools described in the Foundation Level syllabus [ISTQB_FL_SYL] are relevant to, and used for, testing in Agile projects. The Foundation Level syllabus groups the tool types into six categories as follows:

Management of testing and tests

- Test management
- Requirements management
- Incident management (defect tracking)
- Configuration management

Static testing

- Review support
- Static analysis
- Modelling

Specific testing needs

- Data quality assessment
- Support for usability testing

Test execution and logging

- Test execution
- Test harness / unit test framework
- Comparators
- Coverage measurement
- Security testing

Performance and monitoring

- Dynamic analysis
- Performance, load and stress testing
- Monitoring

Not all tools are used the same way within Agile projects as they are in traditional projects, and some tools have different relevance for Agile projects. For example, although the test management tools, requirements management tools, and incident management tools (defect

tracking tools) can be used by Agile teams, some Agile teams opt for an all-inclusive tool (e.g., application lifecycle management or task management) that provides features relevant to Agile development, such as task boards, burndown charts, and user stories.

On the other hand, static analysis and unit test frameworks are of particular relevance to Agile projects because of the emphasis on early testing. Configuration management tools are important to testers in Agile teams due to the high number of automated tests at all levels and the need to store and manage the associated automated test artefacts.

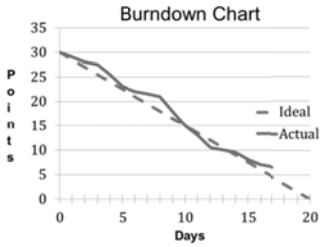
In addition to the tools described in the Foundation Level syllabus [ISTQB_FL_SYL], testers on Agile projects may also use the tools described in the following subsections. These tools are used by the whole team to support team collaboration and information sharing, which are key to Agile practices.

3.4.1 Task Management and Tracking Tools

Agile teams may use physical story/task boards (e.g., whiteboard, corkboard) to manage and track user stories, tests, and other tasks throughout each iteration. However, some Agile teams use application lifecycle management and task management software, including electronic task boards.

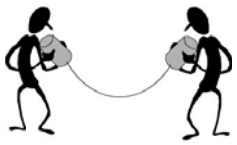
These tools serve the following purposes:

- Record stories and their relevant development and test tasks, to ensure that nothing gets lost during an iteration.
- Capture team members' estimates on their tasks and automatically calculate the effort required to implement a story, supporting efficient iteration planning sessions.
- Associate development tasks and test tasks with the same story, to provide a complete picture of the team's effort required to implement the story.
- Aggregate developer and tester updates to the task status as they complete their work, automatically providing a current calculated snapshot of the status of each story, the iteration, and the overall release.
- Provide a visual representation (via metrics, charts, and dashboards) of the current state of each user story, the iteration, and the release, allowing all stakeholders, including people on geographically distributed teams, to quickly check status.
- Integrate with configuration management tools, which can allow automated recording of code check-ins and builds against tasks, and, in some cases, automated status updates for tasks.



3.4.2 Communication and Information Sharing Tools

In addition to e-mail, documents, and verbal communication, Agile teams often use three additional types of tool to support communication and information sharing: wikis, instant messaging, and desktop sharing.



Wikis allow teams to build and share an online knowledge base on various aspects of the project, including:

- Product feature diagrams, feature discussions, prototype diagrams, photos of whiteboard discussions, and other information.
- Tools and/or techniques for developing and testing found to be useful by other members of the team.
- Metrics, charts, and dashboards on product status, which are particularly useful when the wiki is integrated with other tools such as the build server and the task management system, since the tool can update product status automatically.
- Conversations between team members, similar to instant messaging and email, but in a way that is shared with everyone else on the team.

Instant messaging, audio teleconferencing, and video chat tools provide the following benefits:

- Allow real-time direct communication between team members, especially distributed teams.
- Involve distributed teams in stand-up meetings.
- Reduce telephone bills by use of voice-over-IP technology, removing cost constraints that could reduce team member communication in distributed teams.

Desktop sharing and capture tools provide the following benefits:

- In distributed teams, product demonstrations, code reviews, and even pairing can occur.
- Capturing product demonstrations at the end of each iteration, which can be posted to the team's wiki.

It is important to note that these tools should be used to complement and extend, not replace, face-to-face communication in Agile teams!

3.4.3 Software Build and Distribution Tools

As discussed earlier in this course, daily build and deployment of software is a key practice in Agile teams. This requires the use of continuous integration tools and build distribution tools. The uses, benefits, and risks of these tools were described earlier in Section 1.2.4 "Continuous Integration" but are reproduced below for convenience.

Organisations using continuous integration typically use build tools to implement continuous quality control. Such tools can run static and dynamic tests, measure and profile performance, extract and format documentation from the source code and facilitate manual quality assurance processes.

This continuous application of quality control aims to improve the quality of the product as well as reduce the time taken to deliver it by replacing the traditional practice of applying quality control after completing all development.

Build tools can be linked to automatic deployment tools, which can fetch the appropriate build from the continuous integration or build server and deploy it into one or more development, test, staging, or even production environments. This reduces the errors and delays associated with relying on specialised staff or programmers to install releases in these environments.

Benefits

Build and distribution tools used to implement continuous integration can bring the following benefits:

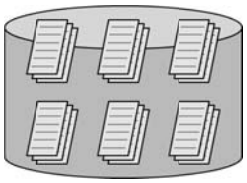
- Allows **earlier detection** and easier root cause analysis of integration problems and conflicting changes.
- Gives the development team **regular feedback** on whether the code is working.
- Keeps the version of the software being tested **within a day** of the version being developed.
- **Reduces regression risk** associated with developer code refactoring due to rapid re-testing of the code base after each small set of changes.
- **Provides confidence** that each day's development work is based on a solid foundation.
- Makes **progress visible**, encouraging developers and testers.
- **Eliminates big-bang integration schedule risks**.
- Provides **constant availability** of executable software throughout the sprint for testing, demonstration, or education purposes.
- **Reduces repetitive manual testing** activities.
- Provides **quick feedback** on decisions made to improve quality and tests.

Risks associated with build and distribution tools include:

- **Tools** have to be introduced and maintained.
- The continuous integration **process** must be defined and established.
- Test automation requires **additional resources** and can be complex to establish.
- Thorough **test coverage** is essential to achieve automated testing advantages.
- Teams sometimes **over-rely on component tests** and perform too little system and acceptance testing.

Continuous integration requires the use of tools, including tools for testing, tools for automating the build process, and tools for version control.

3.4.4 Configuration Management Tools



On Agile teams, configuration management tools may be used not only to store source code and automated tests, but manual tests and other test work products as well. Storing these other artefacts in the same repository as the product source code provides traceability between which versions of the software were tested with which particular versions of the tests, and allows for rapid change without losing historical information.

The main types of version control systems include centralized source control systems and distributed version control systems. The team size, structure, location, and requirements to integrate with other tools will determine which version control system is right for a particular Agile project.

3.4.5 Test Design, Implementation, and Execution Tools

Some tools are useful to Agile testers at specific points in the software testing process. While most of these tools are not new or specific to Agile, they provide important capabilities given the rapid change inherent in Agile projects.

- Test design tools

  Use of tools such as mind maps have become more popular to support quick design and definition of tests for a new feature.

- Test case management tools

  The type of test case management tools used in Agile may be part of the whole team's application lifecycle management or task management tool.

- Test data preparation and generation tools

  Tools that generate data to populate an application's database are very beneficial when a lot of data and combinations of data are necessary to test the application. These tools can also help re-define the database structure as the product undergoes changes during an Agile project. Refactoring the scripts that generate the data becomes part of the technical debt in the project and allows quick updating of test data as changes occur.

  Some test data preparation tools use production data sources as a raw material and use scripts to remove or anonymize sensitive data. Other test data preparation tools can help with validating large data inputs or outputs.

- Test data load tools

  After data has been generated for testing, it needs to be loaded into the application. Manual data entry is often time consuming and error prone, but data load tools are available to make the process reliable and efficient. In fact, many of the data

generator tools include an integrated data load component. In other cases, bulk-loading using the database management systems is also possible.

- Automated test execution tools

  There are test execution tools that are more aligned to Agile testing. Specific tools are available via both commercial and open source avenues to support test first approaches, such as behaviour-driven development, acceptance test-driven development and test-driven development. These tools allow testers and business staff to express the expected system behaviour in tables or natural language using domain specific keywords.

- Exploratory test tools

  Tools that capture and log activities performed on an application during an exploratory testing session are beneficial to the tester and developer, as they record the actions taken. This is useful when a defect is found, as the actions taken before the failure occurred have been captured and can be used to report the defect to the developers efficiently and accurately. Logging steps performed in an exploratory test session may prove to be beneficial if the test is ultimately automated for inclusion in the regression test suite.

3.4.6 Cloud Computing and Virtualization Tools

Virtualization allows a single physical resource (server) to operate as many separate, smaller resources. When virtual machines or cloud instances are used, teams effectively have a greater number of servers available to them for development and testing. This can help to avoid delays associated with waiting for physical servers.

Provisioning a new server or restoring a server is more efficient with the snapshot capabilities that are built into most virtualization tools. Some test management tools now use virtualization technologies to snapshot servers at the point when a failure is detected, allowing testers to share the snapshot with the developers investigating the failure.

3.5 Summary

BDD, ATDD and TDD

Test-driven development (TDD) is an approach to developing code that is driven by automated unit tests: code is written to pass the unit tests. This ensures no code exists that has not been tested (at the unit level) and the automated unit tests provide a means of quickly regression testing the code after code changes for new or changed features and resolving technical debt.

Acceptance test-driven development (ATDD) takes the test-first concept from TDD and applies it to the system testing level. Acceptance criteria are defined using user stories as a basis along with discussion between the team members and the business representatives. From the acceptance criteria, acceptance tests are produced using a natural language form that the customers understand. The code is written to pass the (automated) acceptance tests which are also used for regression testing.

Behaviour-driven development (BDD) is an approach much like ATDD and may be used as an alternative to ATDD. BDD encourages developers to test the code based on the expected behaviour of the software rather than on implementation details.

The Test Pyramid

The test automation pyramid depicts levels of testing, with the unit tests at the bottom, acceptance tests in the middle and GUI tests at the top. The size (width) of the levels

decreases as we move from the bottom to the top and this represents the number of automated tests at each level.

The Testing Quadrants

The Agile testing quadrants model aligns the test levels with the appropriate test types in the Agile methodology. It provides a way to differentiate and describe the types of tests to all stakeholders.

Test types and levels are divided into business facing versus technology facing, and also supporting the team versus critiquing the product.

The Role of the Tester

The Agile team works together in collaboration, making decisions as a team and taking responsibility for quality as a team. Individuals within the team share their skills and knowledge. The team is self-organising and empowered to identify and resolve their own problems. Progress is transparent. In addition testers on the team must question the products behaviour, ensure the testing strategy is credible and be open to feedback, learning from both successes and failures.

During the first iteration (iteration zero in Scrum) the tester collaborates with the rest of the team to identify the scope of the project, plan and install required tools, perform quality risk analysis, define metrics, define ‘done’ and create the task board.

The tester must ensure the integration strategy includes appropriate testing to support frequent deliveries to the customer. Test planning must address issues such as the scope of testing, the test environment, the when and where of testing (including dependencies), and the quality risks to be addressed.

Testers may also benefit from the Agile practices of paired testing, incremental test design and mind mapping.

Assessing quality risks

Using a risk-based testing strategy helps with the proper selection, allocation and prioritization of tests. Quality (or product) risks are specified as the probability of the product failing and impact on the customer if it does. Risks should be identified and assessed during release and iteration planning meetings but a re-evaluation of risks should be undertaken frequently. Testers design tests to mitigate the risks.

Static testing, particularly reviews, may be used to mitigate some risks.

Estimating testing effort

Typically, the work required to implement a user story is estimated in terms of a number of story points. These are relative measures unique to each Agile team and may be equated to effort days or some other meaningful metric.

A common approach to estimation is called planning poker. This is a collaborative approach in which the estimators (not necessarily the whole team) choose a number typically from the Fibonacci sequence. They do this after hearing the product owner (business representative) explain the user story and answer the team’s questions. If everyone has the same estimate, that is used. If different estimates are given the team repeats the estimation after a discussion of the differences. The product owner must not contribute estimates.

Acceptance Criteria, Adequate Coverage, and Other Information for Testing

Both functional and non-functional requirements are expressed by user stories together with acceptance criteria and conversations. These are the main basis for acceptance tests. However, other sources of information may be used as the test basis, including experience, existing product features and characteristics, code, user profiles, quality risks, defect information and relevant standards.

When identifying acceptance criteria the following topics should be considered: functionality, quality characteristics, scenarios, business rules, external interfaces, constraints and data definitions.

The definition of 'done' must be agreed by the team for each test level, user story, feature, iteration and release.

Applying acceptance test-driven development (ATDD)

ATDD involves the following steps:

- identify acceptance criteria
- write acceptance tests
- automate acceptance tests
- code (typically with TDD approach)
- run acceptance tests
- refactor acceptance tests
- customer acceptance

Functional and non-functional black-box test design

Black-box test design techniques described on the Foundation course (equivalence partitioning, boundary value analysis, decision tables, and state transition testing) are applicable to Agile. Developers may use these (and other) techniques when designing unit tests. Testers may use the techniques when creating acceptance tests.

Exploratory testing

Exploratory testing is an approach or mind-set, not a specific testing technique. It requires good test design (using appropriate methods and techniques), careful observation (of the software behaviour), critical thinking (reacting with reasoned actions), diverse ideas (going beyond the obvious) and rich resources (tools, techniques, data and information sources). It is particularly important in Agile projects due to the limited time available for test analysis and the limited details of the user stories.

A test charter guides an exploratory testing time-boxed session of between 60 and 120 minutes. During the session the tester makes notes on observations and failures.

Tools in Agile projects

All the tools described in the Foundation course may be used in Agile projects though some have more relevance than others. Other tools with specific support for Agile methods include the following:

- task management and tracking tools;
- communication and information sharing tools;
- software build and distribution tools;
- configuration management tools;
- test design, implementation, and execution tools;
- cloud computing and virtualization tools.

Agile Values and Principles Exercise 1_1

Learning Objective

FA-1.1.1 K1 Recall the basic concept of Agile software development based on the Agile Manifesto.

The Agile manifesto describes 4 values that are supported by 12 principles. In the table below, match each of the Agile principles onto one or more of the Agile values by putting an “X” in the corresponding table cell.

	Values			
Principles	**Individuals & Interactions**	**Working Software**	**Customer Collaboration**	**Responding to Change**
Early & continuous delivery of valuable software				
Welcome changing requirements				
Deliver working software frequently				
Work together daily				
Trust them to get the job done				
Face to face conversation				
Working software primary measure of progress				
Sustainable development				
Simplicity				
Self-organising teams				
Team reflects and adjusts				

User Stories Exercise 1_2

Learning Objective

FA-1.2.2 K3 Write testable user stories in collaboration with developers and business representatives.

Scenario

A hotel group reservation system allows its customers to search for availability and pricing information on its hotels in the group. Customers can then make a reservation for which they subsequently receive an email confirmation. For all bookings, customers also have to pay a deposit of 20% at the time of booking.

The hotel group runs a premium customer club for regular customers and the premium club members receive promotional offers from time to time amongst other benefits.

Customers may cancel reservations. For premium customers there is no cancellation charge. For non-premium customers there is no charge if cancellation is made prior to 7 calendar days before the arrival date (in which case the full deposit is reaccredited to the customer's credit card). If non-premium customers cancel within 7 days they lose their deposit. All cancellations are emailed to the customers with the relevant information. The relevant hotel must also be notified by email within 10 minutes of any cancellation.

Step 1

In each team, assume you are collaborating as a whole Scrum team of developers, testers and business representatives. The tutor can act as the key Product Owner / business representative for any requirements related queries where you don't want to make your own assumptions. Create a single large story (epic) for the entire cancellation feature. You may use any user story structure you like but if you would like to follow a particular structure here is one that we will use in the solution:

As a	*user*
I want	*some function/action*
So that	*I can achieve some benefit/desired outcome*

In particular ensure you include some testable acceptance criteria for this story / epic.

Step 2

Assume that the Scrum development team (developers, testers, etc.) estimate this story to be too large (i.e. an epic) and that, therefore, it needs to be broken down into smaller more manageable stories. Identify two different ways in which you might consider breaking this epic down into smaller stories and create some examples.

User Stories

Exercise 1_2

Step 1

Story Narrative	Short Name
As a	Role
I want	Something
So that	Benefits

Front of card

Acceptance criteria

Tester in Scrum Team Exercise 3_1

Learning Objective

FA-3.1.4 K3 For a given Agile project, practice the role of a tester in a Scrum team.

Part A

In teams, briefly discuss each of the organisational / behavioural best practices of Scrum teams listed below and identify one point for each that defines the whole team approach to testing and / or how you as a tester should behave / contribute as part of a Scrum team:

Scrum Best Practice	**Whole team approach or tester behaviour/contribution**
Cross-functional	
Self-organising	
Co-located	
Collaborative	
Empowered	
Committed	
Transparent	
Credible	
Open to feedback	
Resilient	

Exercise 3_1 Tester in Scrum Team

Part B

‘Sprint zero’ is the first iteration of the project where many preparation activities take place. Identify 4 activities in which you as a tester should collaborate with the team during this iteration.

Assess Quality Risks — Exercise 3_2

Learning Objective

FA-3.2.1 K3 Assess quality risks within an Agile project.

Consider a simple online bookshop with the following six user stories (not a complete set), sufficient for either a large iteration or a small release. Identify three or four quality risks for each story and then assess each risk and assign a likelihood and impact (using low, medium or high). Then calculate the risk level by multiplying the numeric scores for likelihood and impact where high = 3, medium = 2, low = 1.

The first story has been completed as an example.

Story 1: As a customer,
I want to add a book to my wish list,
so that it is easier to buy later.

Risks:	Likelihood	Impact	Level
• Unable to add a book to the wish list. • Wrong book added to wish list. • Book added to wrong wish list.	Low Low Low	Low Low Low	1 1 1

Story 2: As a customer,
I want to add my credit card details to my account,
so that I can pay for purchases.

Risks:	Likelihood	Impact	Level
•			
•			
•			
•			

Story 3: As a customer,
I want to add a book to my shopping basket,
so that I can buy it later.

Risks:	Likelihood	Impact	Level
•			
•			
•			
•			

Exercise 3_2 Assess Quality Risks

Story 4: As a customer, I want to check-out my shopping basket, so that the items are delivered.			
Risks:	Likelihood	Impact	Level
• • • •			

Story 5: As a customer, I want to browse the online catalogue, so that I can find books of interest.			
Risks:	Likelihood	Impact	Level
• • • •			

Story 6: As a customer, I want to search for specific books, so that I can find specific books of interest.			
Risks:	Likelihood	Impact	Level
• • • •			

Estimate Testing Effort — Exercise 3_3

Learning Objective

FA-3.2.2 K3 Estimate testing effort based on iteration content and quality risks.

Working together in teams, perform a planning poker session (using the cards provided) to estimate the testing effort in terms of a number of story points for each of the following stories. The stories are the same as those used in the previous exercise so take into consideration the risk level previously identified.

Make comments to indicate why a story differs from one or more of the others.

Story	Estimated Story Points	Comments
1 As a customer, I want to add a book to my wish list, so that it is easier to buy later.		
2 As a customer, I want to add my credit card details to my account, so that I can pay for purchases.		
3 As a customer, I want to add a book to my shopping basket, so that I can buy it later.		
4 As a customer, I want to check-out my shopping basket, so that the items are delivered.		
5 As a customer, I want to browse the online catalogue, so that I can find books of interest.		
6 As a customer, I want to search for specific books, so that I can find specific books of interest.		

1 1 1	2 2 2	3 3 3
5 5 5	8 8 8	13 13 13
21 21 21	34 34 34	55 55 55

ATDD Test Cases Exercise 3_5

Learning Objective

FA-3.3.3 K3 Given a user story, write acceptance test-driven development test cases.

Working in teams and considering the user story given below, create as many acceptance test-driven development test cases as you can in the time allowed. Consider each of the following areas:

- basic examples and open questions
- positive (correct behaviour)
- negative paths (exceptions and error conditions)
- non-functional attributes

Specify the test cases using the following ATDD format:

Given: Preconditions

When: Inputs

Then: Outputs

The course tutor will act as the Product Owner for any questions that you may have to better understand the requirements.

User story:	"As a driver, I want the cruise control system to manage the selected speed, so that I can concentrate on other aspects of driving."

Black Box Test Cases Exercise 3_6

Learning Objective

FA-3.3.4 K3 For both functional and non-functional behaviour, write test cases using black box test design techniques based on given user stories.

Part 1

Working in teams consider each of the following user stories and identify one or more suitable black-box test design technique(s). Select from this list of black-box design techniques:

- equivalence partitioning
- boundary value analysis
- decision tables
- state transition testing

You may suggest other test design techniques that you are familiar with.

User stories

1. “As a school trip organizer, I want the trip organization system to assign children from the correct year to the relevant school trips, so that all children are sent on the most appropriate trip.”
2. “As an owner of a health club, I want the subscriptions system to offer discounts to members based on their length of membership, so that more of them are retained as members.”
3. “As a university admissions officer, I want the admissions system to process admissions against the agreed rules, so that I am left with only making subjective decisions based on their interviews.”
4. “As a late sleeper, I want my computerised alarm clock to provide me with three modes for waking, so that I am awakened in the manner that I choose.”
5. “As an air traffic controller, I want the flight management system to display tracked aircraft at least every 4 seconds, so that I am continually aware of their positions.”

Part 2

Create four user stories, one for each of the black-box test design techniques listed above.

Exploratory Testing Exercise 3_7

Learning Objective

FA-3.3.5 K3 Perform exploratory testing to support the testing of an Agile project.

Complete the Test Charter (below) by planning an exploratory testing session for the following user story:

US1_cable_reg_email:

> As a registered user of XYZ cable TV,
> I want to update my registered email address,
> so that I am able to receive emails after my email address changes."

Test charter template

Actor	*intended system user(s)*
Purpose	*what is being tested (i.e. test conditions)*
Setup	*preconditions to test*
Priority	*relative to other charters (based on user story priority/risk)*
Reference	*e.g. user story, risk id.*
Data	*test data*

Exercise 3_7 Exploratory Testing

Activities	*user activities (positive and negative)*
Variations	*in addition to above activities*
Oracle notes	*how to evaluate product correctness (e.g. recording and later comparison)*

1 The Agile Manifesto has 4 statements of values. Match the agile value on the left (1-4) with its traditional counterpart on the right (i-iv).

1) Customer collaboration over	i) Processes and tools
2) Responding to change over	ii) Following a plan
3) Individuals and interactions over	iii) Contract negotiation
4) Working software over	iv) Comprehensive documentation

A 1 – iii, 2 – iv, 3 – ii, 4 – i
B 1 – iii, 2 – ii, 3 – i, 4 – iv
C 1 – iv, 2 – ii, 3 – i, 4 – iii
D 1 – ii, 2 – iii, 3 – iv, 4 – i

K1 (1 mark)

2 Which of the following statements best reflects one of the values of the Agile Manifesto?

A Working software allows the customer to provide rapid feedback to the developer.
B Developers should use unit testing tools to support the testing process.
C Business representatives should provide a backlog of user stories and their estimates to the team.
D Adopting plans to change adds no real value to an agile project.

K1 (1 mark)

3 Which TWO activities below best represent responsibilities that are consistent with agile development's Whole Team approach? Select TWO options.

A Testers are responsible for developing unit tests which they pass on to the developers for testing.
B Business representatives are expected to select the tools the team will use during the project.
C Testers are expected to work with customer representatives to create acceptance tests.
D The whole team, not just testers, has responsibility for the quality of the product.
E Developers are expected to test non-functional requirements (performance, usability, security, etc.)

K2 (1 mark)

4 Which of the following is an advantage of having the whole team responsible for quality?

A Companies no longer need to recruit and train software testing specialists.
B Test automation tasks are now the responsibility of the development team instead of the test team.
C Role barriers are eliminated and team members contribute to project success based on their unique skills and perspectives.
D Project costs are lower because the need for a specialized test team is eliminated.

K2 (1 mark)

5 Which TWO of the following statements are true?

1) Early feedback gives the developers more time to develop new system features because they spend less time reworking features expected in a given iteration.

2) Early feedback enables agile teams to deliver features with the highest business value first, because the customer maintains focus on features with the highest system value.

3) Early feedback reduces costs because it decreases the amount of time needed for system testing.

4) Early feedback makes it more likely that the system built is what the customer wanted because they are given the opportunity to make changes throughout the iteration.

A 1 and 4
B 2 and 3
C 2 and 4
D 1 and 3

K2 (1 mark)

6 Which of the following is a benefit of the agile process promoting early and frequent feedback?

A The total number of defects found during the project is much higher than on traditional software development projects such as waterfall.
B There is less rework because customers see the product regularly.
C It is easy to determine the developer who introduces the most defects when integrating code.
D There is enough time to complete all features scheduled for the given iteration.

K2 (1 mark)

7 Match the following agile software development approaches (numbered 1 to 3) with their corresponding descriptions (numbered I to III).

1) Extreme Programming
2) Scrum
3) Kanban

I. Embraces 5 values to guide development: Communication, Simplicity, Feedback, Courage, and Respect.
II. Divides the project into short iterations called sprints.
III. Optimizes the 'flow' of work in a value-added chain.

A 1-i, 2-iii, 3-ii
B 1-i, 2-ii, 3-iii
C 1-i, 2-ii, 3-iii
D 1-iii, 2-ii, 3-i

K1 (1 mark)

8 During an iteration planning meeting, the team is sharing their thoughts about a user story. The product owner advises that the customer should have one screen to enter information. The developer explains that there are technical limitations for the feature, due to the amount of information needed to be captured on the screen. Another developer says that there are risks about performance as the information will be stored in an external offsite database.

Which of the following would best represent a tester's contribution to this discussion?

A The tester advises that the screen for the user story needs to be a single page to reduce test automation effort.
B The tester advises that usability is more important than performance.
C The tester advises that performance acceptance criteria should standard maximum of 1 second for data storage.
D The tester advises that the user story needs acceptance criteria to be testable.

K3 (1 mark)

9 Which of the following BEST describes a tester participating in a retrospective meeting?

A As a tester participating in a retrospective meeting, I should bring in topics that are related to testing only. All other topics will be covered by different participants.
B As a tester, I participate in a retrospective meeting as an observer, ensuring that the meeting follows the retrospective rules and agile values.
C As a tester participating in a retrospective meeting, I should provide feedback and input on all activities conducted by the team during the sprint.
D As a tester, I should only attend and participate in a retrospective meeting if I have any feedback and input related to activities conducted by the team during the sprint.

K2 (1 mark)

10 Which of the following items should NOT be raised during a retrospective meeting?

A There should be more emphasis on unit testing in the future, to improve overall quality.
B The build process is manual and takes too long. Research and implementation of an automated build framework should be done.
C Tester XYZ is struggling to find defects. Test design training is required for this resource.
D Automated regression test suites are taking too long to run. A review of the tests, to eliminate redundant or unnecessary tests, is required.

K2 (1 mark)

11 Which of the following is NOT a principle of continuous integration?

A Continuous integration helps to build changed software regularly, including testing and deploying, in an automated way.
B Continuous integration allows new builds to be available frequently to testers and stakeholders.
C Continuous integration helps to identify new integration defects early and makes the analysis of these defects easier.
D Continuous integration ensures that testing of builds is done manually, as this generates more reliable results than automated scripts.

K2 (1 mark)

12 Which of the following activities would a tester do during release planning?

A Produce a list of acceptance tests for user stories.
B Help break down user stories into smaller and more detailed tasks.
C Estimate testing tasks generated by new features planned for this iteration.
D Support the clarification of the user stories and ensure that they are testable.

K1 (1 mark)

13 Which of the following is the most appropriate explanation of a 'user story'?

A An artefact that the tester must review and sign off before testing can begin.
B An artefact used to detail only the functional requirements of the system.
C An artefact documented by business representatives to help developers and testers understand the system requirements.
D An artefact written collaboratively by developers, testers, and business representatives to capture requirements.

K1 (1 mark)

14 Which of the following is NOT one of the four statements of values in the Agile Manifesto?

A Individuals and interactions *over* processes and tools
B Working software *over* comprehensive documentation
C Contract resolution *over* customer conciliation
D Responding to change *over* following a plan

K1 (1 mark)

15 Which ONE of the following is an example of the whole-team approach?

A. The whole team determine which features are required in the product.
B. Testers take responsibility for ensuring quality levels are achieved for the whole team.
C. Daily stand-up meetings comprise representatives of testers, developers and the business.
D. Testers enhance the testing performed by business representatives by sharing their test expertise with them.

K2 (1 mark)

16 Which of the following is a benefit of early feedback to the customer rather than the frequent feedback afforded by short iterations?

A Quicker process improvement.
B Faster delivery of key features.
C Better estimation.
D Maintenance of a steady team velocity.

K2 (1 mark)

17 Which of the following statements is true of Scrum?

A It uses time-boxing to synchronize tasks within a sprint.
B It dictates specific software development techniques, such as test first programming.
C It defines four roles: Product Owner, Scrum Master, Developer & Tester.
D It requires deliverables to be released when ready rather than at the end of a sprint.

K1 (1 mark)

18 Given the following user story descriptions:

i. "As a member secretary, I want to be able to check whose subscriptions have been paid."
ii. "As a currency trader, I want large fluctuations to be highlighted, so that I can either buy or dump currency to make a profit."
iii. "As a supermarket buyer, I want the system to ensure fresh stock is always on the shelves at the right price, so that I can concentrate on marketing."
iv. "As a car driver, I want the car to maintain a safe distance to the car in front based on provided car, road and weather conditions, so that I avoid unnecessary collisions."

Which of the following BEST matches likely areas of concern with these story descriptions if using the INVEST technique?

A i – V; ii – T; iii – S; iv – I
B i – T; ii – S; iii – N; iv – E
C i – E; ii – V; iii – T; iv – N
D i – S; ii – I; iii – N; iv – V

K3 (1 mark)

19 Which of the following statements is MOST likely to be made by a participant of a retrospective?

A "Each participant was allocated a specific area (e.g. process, people, tools, testing, or programming) and we each identified how this area could be improved."
B "The participants comprised the Scrum Master, developers and testers and we jointly created a report on improvements for approval by the business representative."
C "We identified everything that we thought worked well, and then determined if we needed to do anything to ensure it would continue to work in the next iteration."
D "Root cause analysis of defects was used to identify the individual responsible for the defect and they now must ensure similar defects do not occur in the future."

K2 (1 mark)

20 Which of the following problems is MOST likely to be addressed by the introduction of continuous integration?

A The agile team are reliant on unit testing as their primary quality assurance technique.
B The deployment of new releases is delayed due to the lack of available specialized staff.
C Static code analysis currently identifies a high number of anomalies in the code.
D Acceptance tests written by business representatives are not automated.

K2 (1 mark)

21 Which of the following is MOST likely to be an activity performed by a tester in release planning rather than in *iteration planning*?

A Estimate the test effort associated with the user stories.
B Identify functional and non-functional aspects of the system to be tested.
C Break user stories into testing tasks.
D Create acceptance tests for user stories.

K1 (1 mark)

1 Which of the following test activities is typically done during agile projects, but is not as common on traditional projects?

A Testers write detailed test plans so all team members can understand what will be tested during each iteration.
B Testers are heavily involved in the creation of automated test cases which are then used to verify the implementation of the requirements.
C Testers perform exploratory testing in order to find important defects quickly.
D Testers collaborate with developers to better understand what needs to be tested.

K2 (1 mark)

2 Consider the following activities:

i. Strict enforcement of system test level entry and exit criteria.
ii. Collaboration between tester, developer, and business stakeholders to define acceptance criteria.
iii. Functional verification testing of user stories developed in the previous iteration.

Which of the following combination of these activities should occur in an agile project?

A ii only
B i and ii
C ii and iii
D iii only

K2 (1 mark)

3 Which TWO of the following statements are true on agile projects? Select TWO options.

A Testers should work closely with developers while retaining an objective outlook.
B Test managers do not exist in organizations doing agile development.
C There is no difference between what testers and developers do on agile projects.
D Developers should rely on testers to create the automated regression tests.
E A selection of users may perform beta testing on the product after the completion of a series of iterations.

K2 (1 mark)

4 Which of the following statements about independent testing on agile projects is FALSE?

A There can be a risk of losing test independence for organizations introducing agile.
B Independent testers will find more defects than developers regardless of test level.
C Independent testing can be introduced at the end of a sprint.
D The independent test team can be part of another team.

K2 (1 mark)

5 In an agile project, which of the following would best denote product quality at the end of iteration 6 of a new system release consisting of 8 iterations?

A No severity 1 or 2 defects were detected during system testing of iteration 6, which allowed the teams to move into iteration 7.
B The results of a customer beta test on the iteration 6 software release indicate that the system works correctly and that it has improved productivity.
C The agile team has been successfully tracking to estimates, with limited variance showing on the burndown charts for all iterations to date.
D All story cards in scope for each iteration, up to the current iteration, have been marked as “Done”, but with some technical debt being incurred.

K2 (1 mark)

6 Which of the following is best at showing the team’s progress against estimates?

A Burndown charts.
B Automation logs.
C The agile task board showing user story and task progress.
D Defect tracking tools.

K2 (1 mark)

7 The business advises during iteration 5 planning that they require changes to the system delivered in iteration 3. Of the following activities, which would need to be done first to minimize the introduction of regression risk when this feature is changed?

A Review and update all manual and automated tests impacted by this change to meet the new acceptance criteria.
B Write new manual and automated tests for the feature and add them to the regression test suite.
C Automate all test cases from the previous iteration and add them to the automated regression test suite.
D Increase the amount of test automation around the system to include more detailed test conditions.

K2 (1 mark)

8 Which TWO of the following are reasons why automation is essential within agile projects?

i. So that teams maintain or increase their velocity.
ii. To prevent the test team from becoming bored with manual, repetitive tasks.
iii. To retest all test cases from previous iterations.
iv. To eliminate regression in the product due to high code churn.
v. To ensure that code changes do not break the software build.

A i and iv
B i and v
C iii and iv
D ii and v

K2 (1 mark)

9 In agile projects there is more need for testers to understand and develop test automation scripts than in traditional projects. Of the following, which are the TWO reasons why this is a necessary skill on agile projects?

i. Requirements change daily and have to be regression tested. This rapid change requires automated tests because manual testing is too slow.
ii. The tests should generate feedback on product quality as early as possible. So all acceptance tests should be executed in each iteration, ideally as modifications are made. In practice that can only be realized by automated tests.
iii. Test-First and Continuous Integration Practice require that the regression test suite is executed whenever changed code is checked-in. In practice that can only be realized by automated tests.
iv. Iterations or sprints are of fixed length. The team has to guarantee that all tests can be completely executed at the last day of each iteration/sprint. In practice, that can only be realized by automated tests.
v. Agile projects rely on unit testing rather than on systems testing. Since unit tests cannot be executed manually, all tests have to be automated tests.

A i & iii
B ii & v
C iv & v
D ii and iii

K2 (1 mark)

10 Which tasks are typically expected of a tester on an agile project?

i. decide on user acceptance.
ii. design, create and execute appropriate tests.
iii. schedule defect reports for analysis.
iv. automate and maintain tests.
v. improve program logic by pair programming.

A i & iii
B ii & iii
C ii & iv
D ii & v

K2 (1 mark)

11 Which of the following is NOT a typical task performed by the tester within an agile team?

A To automate tests and maintain them.
B To mentor and coach other team members.
C To produce and update burndown charts.
D To participate in code analysing activities.

K2 (1 mark)

12 The term “burndown” refers to which of the following?

A A chart showing which team members are working the most, and are likely to be under stress.
B A chart showing the progress of each user story, and when they are likely to be completed.
C A chart showing the amount of work left to be done, versus the time allocated for the iteration.
D A chart showing defects that have been fixed, and when the remaining defects are likely to be fixed.

K1 (1 mark)

13 Given the following statements made by testers, which is the MOST likely to be working on a traditional project (rather than an agile project)?

A “Most of my manual testing is exploratory testing and error guessing”.
B “I often pair up with a developer to do my testing”.
C “I coach business representatives on writing acceptance tests”.
D “Much of my time is spent doing manual integration testing.”

K2 (1 mark)

14 Which of the following is LEAST likely to describe how testing activities are performed on Agile projects?

A Regression testing typically occurs at different test levels throughout the iteration
B Business representatives test stories during implementation and not just prior to release
C Automated regression tests are performed as part of continuous integration
D Pairing involves developer-tester pairs separately developing and testing the same feature

K2 (1 mark)

15 When a semi-independent test team is used on an agile project, the testers are loaned long-term to the agile team. Which of the following BEST describes the BENEFITS of such an approach?

A The permanent association of the testers with an external test team allows them to maintain their independence
B As the testers move from team to team they are not concerned with product features so instead concentrate on automated regression testing
C The tester’s involvement in the external test team means they don’t feel the need to build relationships with the developers and business representatives
D As the testers work on the whole iteration they have time to code features during the first half of the iteration

K2 (1 mark)

16 Which of the following are BEST suited to communicating test progress?

A Defects found and fixed & Burndown chart
B Code churn & Agile taskboard
C Customer satisfaction surveys & Defects found and fixed
D Burndown chart & Agile taskboard

K2 (1 mark)

17 Which of the following statements about regression testing on agile projects is INCORRECT?

A The automation of regression tests is needed to provide fast feedback on the quality of existing product features.
B On each iteration, testers should determine which test cases should be added to the regression test suite, and which should be removed.
C When a regression test fails due to a product defect, the defect should be fixed and the regression test removed from the regression test suite.
D Each time a new build is deployed a set of critical automated regression tests should be run to provide feedback on the build quality.

K2 (1 mark)

18 Which TWO of the following skills are MOST likely to be NEW to a test analyst moving from a traditional project to an agile project?

A Be positive and solution-oriented with team members and stakeholders.
B Display critical, quality-oriented, sceptical thinking about the product.
C Actively acquire information from stakeholders (rather than written specifications).
D Accurately evaluate and report test results, test progress, and product quality.
E Plan and organize their own work.

K2 (1 mark)

19 Which TWO of the following activities are MOST likely to be NEW to a test analyst moving from a traditional project to an agile project?

A Understand, implement, and update the test strategy.
B Measure and report test coverage.
C Configure, use, and manage test environments and test data.
D Report defects and work with the team to resolve them.
E Coach other team members in testing.

K2 (1 mark)

1 Which of the following statements about Test Driven Development (TDD) is FALSE?

A TDD is a ”test first” approach to develop reusable automated tests.
B The TDD cycle is continuously used until the software product is released.
C TDD helps to document the code for future maintenance efforts.
D The result of TDD are test classes used by the developer to develop test cases.

K1 (1 mark)

2 What does the term ‘Test Pyramid’ refer to and illustrate situations for?

A The team’s testing workload increases from sprint to sprint.
B The backlog size, and thus the number of tests, decreases.
C The number of automated unit tests is higher than the number of automated tests for higher test levels.
D The number of automated tests in place increases from sprint to sprint.

K1 (1 mark)

3 Which of the following demonstrates effective use of the testing quadrants?

A When communicating test ideas, the tester can refer to the matching test quadrant, so that the rest of the team will better understand the purpose of the test.
B The tester can use the types of tests described in the testing quadrants as a coverage metric, the more tests covered from each quadrant, the higher the test coverage.
C The team should pick a number of tests expected from each quadrant, and the tester should design and execute those tests to ensure all levels and types of tests have been executed.
D The tester can use the testing quadrants during risk analysis; with the lower level quadrants representing lower risk to customer.

K2 (1 mark)

4 Given the following user stories:

> “As a bank teller, I can easily navigate through the system menu and links, and find the information I am looking for”
> “For all users, the system must display all queries in less than 2 seconds, 90% of the time”

And the associated test cases:

> TC1: Login as bank teller. Enter customer ID. Verify that the customer transaction history is easy to find, and that navigating through the menus is intuitive.
> TC2: Login as bank teller: Enter customer Name. Verify that the customer accounts are easy to find and that navigating through the menus is intuitive.
> TC3: Simulate expected traffic on system and validate the time for customer transaction history to display is less than 2 seconds.

Which TWO test quadrants would the above test cases be part of?

A Q1 unit level, technology facing & Q2 system level, business facing.
B Q2 system level, business facing & Q3 system or user acceptance level, business facing.
C Q3 system or user acceptance level, business facing & Q4 system or operation acceptance level, technology facing.
D Q2 system level, business facing & Q4 system or operation acceptance level, technology facing.

K2 (1 mark)

5 At the beginning of the 5th iteration of a project, a new requirement was introduced to support a new type of browser. The tester realizes that the existing test automation framework and scripts will not support the new type of browser. What is the best course of action for the tester on this team to take?

A The tester should notify the team that they are planning on working extra hours throughout the next 2 sprints in order to update the existing test automation framework and scripts to support the new type of browser so as not to disturb the existing sprint plan.
B The tester will notify the team of the issue. A risk analysis is done, and the team decides that regression testing must be performed on the new type of browser in addition to the other supported browsers. The tester will update the sprint plan by adding tasks to modify the framework and scripts to support the new type of browser.
C The tester does some research and concludes that the risk is low that any new defects would be introduced in the new type of browser that have not already been found in other supported browsers. The tester continues with the existing sprint plan and makes no changes to test automation framework or scripts.
D The tester will stop what they are doing, design specific tests for compatibility testing of the new type of browser, and communicate with the team that any other testing work for the sprint will have to be pushed to the next iteration.

K2 (1 mark)

6 Given the following results from a product risk analysis that occurred at the beginning of an iteration:

- User story 1 (Performance): likelihood: high, impact: high
- User story 2 (Security): likelihood: high, impact: high
- User story 3 (Functional): likelihood: medium, impact: high
- User story 4 (Functional): likelihood: high, impact: medium
- User story 5 (Compatibility): likelihood: low, impact: low
- User story 6 (Recoverability): likelihood: low, impact: low

Which TWO of the following describes best what the team should do with this information? Select TWO options.

A Move onto planning poker session to estimate effort for user stories, and determine what can be done in the current iteration, and what needs to be added to backlog.
B Remove user stories 5 and 6 from the current iteration and move to a later iteration.
C Because of the number of high likelihood, high impact risks slotted for this iteration, the team has no choice but to extend the timeframe of the iteration by 2 weeks.
D The team should collaborate on effective ways to mitigate the high likelihood, high impact risks.
E The team should plan to complete all items in the current sprint, but save the lower risk items for the end of the sprint, and only test these items if there is time.

K3 (1 mark)

7 Which of the following statements is FALSE with respect to exploratory testing?

A Exploratory testing encompasses concurrent learning, test design, and execution.
B Exploratory testing eliminates the need for testers to prepare test ideas prior to test execution.
C Best results are achieved when exploratory testing is combined with other test strategies.
D Exploratory testers need to have a solid understanding of the system under test.

K1 (1 mark)

8 Given the following attributes related to test-driven development, acceptance test-driven development and behaviour-driven development, which TWO of the following are common to ALL three approaches?

A The derived tests are used as the basis for continuous integration.
B The tests are defined before the code is written using a test-first approach.
C The approach is supported by tools and provides test automation.
D The approach supports high-level testing, such as system and acceptance testing.
E The approach ensures the derived tests are easy-to-understand and review.

K1 (1 mark)

9 Which of the following CORRECTLY describes the concepts presented by the ‘test pyramid’?

A. There should be more acceptance tests than tests at other test levels as these are closer to the user experience.
B. The tests developed for the unit and integration test levels should normally be automated.
C. The automated tests developed for system and acceptance testing are normally based on API-based tools.
D. There should be more integration tests than unit tests due to the use of continuous integration..

K1 (1 mark)

10 Which of the following CORRECTLY describes the testing on an agile project as described by the 'testing quadrants'?

A The testing associated with the 'System or Operational Acceptance Level' quadrant includes most of the non-functional test types.
B Tests from all four quadrants should be planned for the testing on an agile project.
C Static testing is included in one of the technology-facing quadrants.
D The testing associated with the technology facing quadrants should be performed ahead of that associated with the business facing quadrants.

K2 (1 mark)

11 The tester collaborates on the following activities as part of Sprint Zero:

i. Plan, acquire, and install needed tools
ii. Create an initial system architecture
iii. Specify the definition of "done"
iv. Define test metrics to measure the test process, the progress of testing in the project, and product quality
v. Create the task board
vi. Identify the scope of the project
vii. Create an initial test strategy for all test levels
viii. Perform an initial quality risk analysis

Which of the following provides the MOST logical order for performing these activities?

A ii – vii – viii – vi – i – v – iv - iii
B v – vi – vii – viii – iv – iii – i - ii
C viii – iii – vii – vi – iv – ii – i - v
D vi – ii – viii – vii – iv – iii – i – v

K3 (1 mark)

12 Five user stories have been selected for the current iteration and the following values determined for the identified quality risks associated with them:

Story1:	likelihood = high;	impact = low
Story2:	likelihood = high;	impact = high
Story3:	likelihood = medium;	impact = medium
Story4:	likelihood = low;	impact = high
Story5:	likelihood = low;	impact = low

Which of the following is the MOST logical order of testing and associated test effort?

A Story2 (8 hrs) – Story3 (6 hrs) – Story4 (4 hrs) – Story1 (4 hrs) – Story5 (2 hrs)
B Story2 (2 hrs) – Story3 (4 hrs) – Story4 (6 hrs) – Story1 (6 hrs) – Story5 (8 hrs)
C Story2 (8 hrs) – Story4 (8 hrs) – Story3 (6 hrs) – Story1 (4 hrs) – Story5 (4 hrs)
D Story5 (8 hrs) – Story1 (6 hrs) – Story4 (4 hrs) – Story3 (4 hrs) – Story2 (2 hrs)

K3 (1 mark)

1 Given the following user story: “As the president, any data I upload should not be viewable by any other user of the system”.

During the first poker planning session, the following story points were given based on risk, effort, complexity, and proper extent of testing:

- Customers: 5
- Developers: 5
- Testers: 20

What is the best outcome following this planning session?

A Because the customer’s and developer’s size estimates match, the team can be confident that this estimate is good and should move onto the next user story.
B The team should hold a conversation to understand why the testers felt this user story was significantly more work. Another round of the planning poker session should occur following that discussion.
C Because the customer owns the system in the end, the customers’ estimates should be taken as correct when there is a conflict.
D The poker planning sessions should continue until all estimated story points are an exact match between customers, developers, and testers.

K3 (1 mark)

2 An agile team is assigned to a project to update an existing medical device to newer technologies. Since the last release of the existing medical device, a new version of the medical device standard has been released. User access to the device is changing and will be documented in user stories.

Based on this information, and in addition to the user stories, which of the following would best provide relevant information to support your testing activities?

i. Updated version of standards document for medical system.
ii. Existing defects or typical defect areas in existing system.
iii. Obsolete user access test cases and results for existing application.
iv. Performance metrics for existing application.
v. Defects logged during other similar conversion projects for medical devices.

A i, ii, iii, iv
B ii, iv, v
C i, ii, v
D All of the above

K3 (1 mark)

3 Which alternative is the BEST description of when to stop testing (release criteria) in an agile project?

A All test cases have been executed.
B The probability of remaining faults has been reduced to a level that can be accepted by the customer.
C The achieved test coverage is considered enough. The coverage limit is justified by the complexity of the included functionality, its implementation, and the risks involved.
D The iteration/sprint is finished.

K2 (1 mark)

4 Which TWO of the following are examples of testable acceptance criteria for test related activities? Select TWO options.

A Structure based testing: White box testing in addition to black box testing is used.
B System testing: At least 80% of functional regression tests are automated.
C Security testing: A threat risk analysis scan is completed with no faults identified.
D Performance testing: The application is responding in a reasonable amount of time with 5000 users.
E Compatibility testing: The application is working on all major browsers.

K2 (1 mark)

5 Given the following User Story: "As a bank teller, I would like to be able to view all of my customer's bank transactions on the screen, so I can answer his/her questions".

Which of the following can be considered as relevant acceptance test cases?

i Login as a bank teller, get the customer's account balance for all open accounts.
ii. Login as a bank teller, enter a customer account ID, get his/her transactions history on the screen.
iii. Login as a bank teller, request customer account ID by using name abbreviations, and get his transaction history on the screen.
iv. Login as a bank teller, enter a customer IBAN (international bank account number), get his transaction history on the screen.
v. Login as a Bank Teller, enter a customer account ID, get the transactions history in less than 3 seconds on screen.

A i, ii, iv
B i, iii, iv
C ii, iv, v
D ii, iii, iv

K3 (1 mark)

6 Given the following user story: "An online application charges customers to ship purchased items based on the following criteria:

- Standard shipping costs for under 6 items.
- Shipping is $5 for 6-10 items.
- Shipping is free for more than 10 items.

Which of the following is the best black box test design technique for the user story?

A State Transition testing: test the following states – browsing, logged in, selecting, purchasing, confirming, and exiting.
B Decision tables: test the following conditions – user logged in; at least 1 item in cart; purchase confirmed; funding approved; with the resulting action of – Ship Item.
C Boundary Value Analysis: test the following inputs – 0, 5, 6, 10, 11, max.
D Use Case Testing: actor=customer; prerequisites=customer logs in, selects and purchases items; post conditions= items are shipped.

K3 (1 mark)

7 Your manager would like to introduce exploratory testing to your agile team. He has received the following suggestions on how to proceed from previous colleagues:

i. User stories are assigned to testers who are completely new to the user story. 120 minutes is allocated to complete exploratory testing on the user story. Testers do not need to document tests, or test results, but do need to log defects if any are encountered.
ii. User stories are assigned to testers who have already completed risk based testing on the same areas. 120 minutes is allocated to complete exploratory testing for this user story. The team expects at the end of the 120 minutes to have a list of test ideas, including data and actors, results and issues encountered, and list of defects to be logged in the defect management tool.
iii. A user story is assigned to a business representative. The business representative is told to use the system like the customer would on a day-to-day basis. If issues are encountered, the business representative is told to inform the tester, so that they can prioritize and log the defect.
iv. A user story is assigned to a tester for exploratory testing. Tester is told to learn the functionality of the user story, to make sure the functionality is correct and to include negative testing. There is no set deadline for this exploratory testing to be complete; it depends on what is found by the tester. Documentation is not necessary, but defects need to be logged in the defect tracking tool.

Your manager presents you with his conclusions about how best to introduce exploratory testing to an agile team. Which one of your manager's conclusions is correct?

A Scenario i IS NOT the best way because: in exploratory testing, test design and test execution happen at the same time but are guided by a documented test charter that includes actors, test conditions, test data, etc. Test results are also documented and will guide the next test.
B Scenario ii IS the best way because: In this case, the testers have knowledge of the user story already, which will help them come up with test conditions and ideas. The team is using time-boxed exploratory test sessions. The team is expected to document test conditions, data, and user information, and to log results of the test. Issues are logged in a defect tracking tool just like any other test technique.
C Scenario iii IS NOT the best way because: This could be describing system acceptance testing, but not exploratory testing.
D Scenario iv IS NOT the best way because: Documentation is necessary for exploratory testing, and testers must log test ideas and results of testing. The results of testing are used to guide future exploratory testing.

K3 (1 mark)

8 Which of the following is one of the purposes of an Application Lifecycle Management (ALM) tool on an agile project?

A An ALM tool allows teams to build up a knowledge base on tools and techniques for development and testing activities.
B An ALM tool provides quick response about the build quality and details about code changes.
C An ALM tool provides visibility into the current state of the application, especially with distributed teams.
D An ALM tool generates and loads large volumes and combinations of data to use for testing.

K1 (1 mark)

9 An agile team are estimating the size of a user story using planning poker and have selected the following five estimates based on their knowledge and the assessed risk score for the story: 8, 13, 13, 21 and 13.

The tester, who selected 21, explains that her score reflects her belief that the testing effort required will be higher than average due to needing to use a specialised test harness. One of the developers, who selected 8 and is most familiar with the story, explains that the feature is smaller than average and not complex. Which of the following sets of estimates would you MOST likely expect for the next round of planning poker:

A 8, 13, 13, 21 and 13
B 13, 21, 21, 34, and 21
C 13, 13, 13, 13 and 13
D 8, 8, 8, 13, and 8

K3 (1 mark)

10 An agile project is creating a new customer management system that complies with recent EU legislation on data privacy (the project is aimed at catching up with competitors and avoiding censure by the authorities). As a result of a retrospective it has been agreed that the tester should update the test strategy for the next release as too many defects had to be fixed after being found by users in the first release. The tester has worked on the project since it began and has access to the product backlog for the next release.

Which of the following would BEST describe the next step for the tester?

A Categorize the defects found so far on the project in a defect taxonomy.
B Determine whether the currently-available test tool support is sufficient and whether they have the skills needed to use these tools.
C Identify the applicable EU data privacy standards to determine compliance with them.
D Collect information on defects from similar products related to the features in the product backlog.

K3 (1 mark)

11 Which of the following is LEAST likely to be a testable acceptance criteria?

A Grid references for locations shall be provided in both the Universal Transverse Mercator (UTM) system and the British national grid reference system.
B Customer due diligence checks shall be applied as described by the current version of the FSA policy statement PS11/15 (Financial crime: a guide for firms).
C The new off-the-shelf PC-based stock pricing system shall use no more than 60% of the available processing power on the user's PC.
D Updates to vehicle positions by the tracking subsystem shall be made no less than once per 8 seconds on the reference architecture.

K2 (1 mark)

12 Given the following user story:

"As a partner in the law firm, I want the billing system to identify our most lucrative clients, so that I can target them with special marketing."

Which of the following is LEAST likely to be used as an acceptance test case?

A Given: 1000 Clients are held on the system; and
100 Clients have provided greater than £50K profit
When: Partner checks for clients providing more than £50K profit
Then: Results shall be presented within 5 seconds of the request
B Given: Associates have been added as users of the system
When: Associate checks for clients providing more than £75K profit
Then: Report that associates are not cleared for this system feature
C Given: 20 Clients are held on the system; and
5 Clients have provided no profit
When: Partner checks for clients providing no profits
Then: Present a list of the 5 clients
D Given: No Clients have been entered on the system
When: Partner checks for clients providing more than £200K profit
Then: Report that no clients have been entered on system

K3 (1 mark)

13 Given the following user story:

"As a driver of a car, I want the locking system to automatically lock the car 1 minute after the ignition is switched off, so the car is locked if I forget."

And the corresponding test cases:

T1: Car unlocked – Turn Ignition off - Wait 70 secs - Check car locked
T2: Car unlocked - Turn Ignition off - Wait 30 secs - Lock car - Wait 40 secs - Check car locked
T3: Car locked - Turn Ignition off - Wait 20 secs - Unlock car - Wait 50 secs - Check car locked
T4: Car locked - Turn Ignition off - Wait 70 secs - Check car is locked

Which of the following test case design techniques is MOST likely to have been used to generate these test cases?

A Equivalence partitioning
B Boundary value analysis
C State transition testing
D Decision table testing

K3 (1 mark)

14 Which of the following descriptions is LEAST likely to apply to the exploratory testing performed as part of an agile project?

A A 30 minutes session was performed to learn how the system worked largely based on the exploratory questions: “What happens if.....?” and “What should happen when.....?”
B The test charter for the open-ended test session required all interesting CRUD (Create, Read, Update and Delete) actions on the member database to be performed.
C A deep coverage session based on a test charter in which the purpose was set out to cover boundaries and corner cases was performed.
D The documentation showed the levels of both risk and test coverage for the analysis session covering the functionality of the application.

K3 (1 mark)

15 Which of the following BEST matches the tool categories used on agile projects with the following attributes?

i. Store manual and automated tests
ii. Link tasks with stories
iii. Use a snapshot to efficiently provision a new server
iv. Visually show story status
v. Load data into the application
vi. Provide a platform for code reviews

A
Task Management and Tracking Tools - iv
Cloud Computing and Virtualization Tools - iii
Configuration Management Tools - i
Test Design, Implementation, and Execution Tools - v
Communication and Information Sharing Tools - vi

B
Task Management and Tracking Tools - i
Cloud Computing and Virtualization Tools - v
Configuration Management Tools - ii
Test Design, Implementation, and Execution Tools - vi
Communication and Information Sharing Tools - iv

C
Task Management and Tracking Tools - vi
Cloud Computing and Virtualization Tools - v
Configuration Management Tools - iii
Test Design, Implementation, and Execution Tools - iv
Communication and Information Sharing Tools - ii

D
Task Management and Tracking Tools - ii
Cloud Computing and Virtualization Tools - vi
Configuration Management Tools - iii
Test Design, Implementation, and Execution Tools - v
Communication and Information Sharing Tools - iv

K1 (1 mark)

Appendix A

References

A.1 Standards

[DO-178B] RTCA/FAA DO-178B
Software Considerations in Airborne Systems and Equipment Certification, 1992.

[ISO25000] ISO/IEC 25000:2005
Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE), 2005

A.2 ISTQB Documents

[ISTQB_AL_OVIEW] ISTQB Advanced Level Overview
Version 1.0

[ISTQB_ALTA_SYL] ISTQB Advanced Level Test Analyst Syllabus
Version 2012

[ISTQB_ALTM_SYL] ISTQB Advanced Level Test Manager Syllabus
Version 2012

[ISTQB_ALTTA_SYL] ISTQB Advanced Level Technical Test Analyst Syllabus
Version 2012

[ISTQB_FA_OVIEW] ISTQB Foundation Level Agile Tester Overview
Version 1.0

[ISTQB_FL_SYL] ISTQB Foundation Level Syllabus
Version 2011

[ISTQB_GLOSSARY] Standard glossary of terms used in Software Testing
Version 2.2, 2012

A.3 Trademarks

The following registered trademarks and service marks are used in this document:

- ISTQB is a registered trademark of the International Software Testing Qualifications Board
- TPI® Next is a registered trademark of Sogeti Nederland B.V.

A.4 Books

[Aalst13] Leo van der Aalst and Cecile Davis
“TMap NEXT® in Scrum”
ICT-Books.com, 2013

[Adzic09] Gojko Adzic
"Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing"
Neuri Limited, 2009

[Anderson13] David Anderson
"Kanban: Successful Evolutionary Change for Your Technology Business"
Blue Hole Press, 2010

[Beck02] Kent Beck
"Test-driven Development: By Example"
Addison-Wesley Professional, 2002

[Beck04] Kent Beck and Cynthia Andres
"Extreme Programming Explained: Embrace Change, 2e"
Addison-Wesley Professional, 2004

[Black07] Rex Black
"Pragmatic Software Testing"
John Wiley and Sons, 2007

[Black09] Rex Black
"Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing, 3e"
Wiley, 2009

[Chelimsky10] David Chelimsky et al
"The RSpec Book: Behavior Driven Development with Rspec, Cucumber, and Friends"
Pragmatic Bookshelf, 2010

[Cohn04] Mike Cohn
"User Stories Applied: For Agile Software Development"
Addison-Wesley Professional, 2004

[Cohn10] Mike Cohn
"Succeeding with Agile: Software Development with Scrum"
Pearson Education, Inc., 2010

[Crispin08] Lisa Crispin and Janet Gregory
"Agile Testing: A Practical Guide for Testers and Agile Teams"
Addison-Wesley Professional, 2008

[Davies01] Rachel Davies
"The Power of Stories"
XP 2001. Sardinia, 2001.

[Goucher09] Adam Goucher and Tim Reilly, editors
"Beautiful Testing: Leading Professionals Reveal How They Improve Software"
O'Reilly Media, 2009

[Jeffries00] Ron Jeffries, Ann Anderson, and Chet Hendrickson
"Extreme Programming Installed"
Addison-Wesley Professional, 2000

[Jones11] Capers Jones and Olivier Bonsignour
"The Economics of Software Quality"
Addison-Wesley Professional, 2011

[Linz14] Tilo Linz
"Testing in Scrum: A Guide for Software Quality Assurance in the Agile World"
Rocky Nook, 2014

[Koskela07] Lasse Koskela
"Test driven: TDD and acceptance TDD for Java developers"
Manning, 2007

[Meszaros07] Gerard Meszaros
"XUnit Test Patterns: Refactoring Test Code"
Addison-Wesley, 2007

[Schwaber01] Ken Schwaber and Mike Beedle
"Agile Software Development with Scrum"
Prentice Hall, 2001

[vanVeenendaal12] Erik van Veenendaal
"The PRISMA approach"
Uitgeverij Tutein Nolthenius, 2012

[Wiegers13] Karl Wiegers and Joy Beatty
"Software Requirements, 3e"
Microsoft Press, 2013.

A.5 Articles

[Larman03] Craig Larman and Victor Basili
"Iterative and Incremental Development: A Brief History"
IEEE Computer, June 2003

A.6 Internet

The following references point to information available on the Internet. These references were active at the time of writing but some may no longer be available.

[Web-1] ISTQB
http://www.istqb.org

[Web-2] Agile Manifesto
http://agilemanifesto.org

[Web-3] The Values of Extreme Programming
http://www.extremeprogramming.org/values.html

[Web-4] An Agile Software Development Resource
http://xprogramming.com

[Web-5] Scrum.org
https://www.scrum.org

[Web-6] Scrum Alliance
www.scrumalliance.org

[Web-7] Kanban kick-start example
http://blog.crisp.se/2009/11/16/henrikkniberg/kanban-kick-start-example

[Web-8] LeanKit
http://leankit.com/kanban/what-is-kanban

[Web-9] Website for exploring Agile and Lean
http://xp123.com/articles/invest-in-good-stories-and-smart-tasks

[Web-10] The Agile Alliance
http://guide.agilealliance.org/guide/threecs.html

[Web-11] Article on User Stories
http://xprogramming.com/articles/expcardconversationconfirmation

[Web-12] Kanban and Scrum marking the most of both
http://www.infoq.com/minibooks/kanban-scrum-minibook

A.7 Other References

The following references point to information available on the Internet and elsewhere. Even though these references were checked at the time of writing they may subsequently become unavailable.

[Agile Alliance Guide] Various contributors
http://guide.Agilealliance.org/

[Agilemanifesto] Various contributors
www.agilemanifesto.org

[Hendrickson1] Elisabeth Hendrickson
“Acceptance Test-driven Development”
testobsessed.com/2008/12/acceptance-test-driven-development-atdd-an-overview

[Hendrickson2] Elisabeth Hendrickson
“Test Heuristics Cheat Sheet”
testobsessed.com/wp-content/uploads/2011/04/testheuristicscheatsheetv1.pdf

[INVEST] Bill Wake
“INVEST in Good Stories, and SMART Tasks”
xp123.com/articles/invest-in-good-stories-and-smart-tasks

[Kubaczkowski] Greg Kubaczkowski and Rex Black
“Mission Made Possible”
www.rbcs-us.com/images/documents/Mission-Made-Possible.pdf

Release Notes

Version 2.4 of July 4th, 2014

This new version has been developed to support the Expert Level Test Automation – Engineer (ETAE) syllabus. Existing terms have not been changed; hence, other syllabi are not affected.

Term	Change Type	Reason
API	Added	Acronym used in ETAE
API testing	Changed	ETAE keyword; Definition reworked.
capture/playback	Added	ETAE keyword
CLI	Added	Acronym used in ETAE
CLI testing	Added	ETAE keyword
EMTE	Added	Acronym used in ETAE
equivalent manual test effort	Added	ETAE keyword
generic test automation architecture	Added	ETAE keyword
GUI	Added	Acronym used in ETAE
GUI testing	Added	ETAE keyword
level of intrusion	Added	ETAE keyword
linear scripting	Added	ETAE keyword
process-driven testing	Added	ETAE keyword
structured scripting	Added	ETAE keyword
SUT	Added	Acronym used in ETAE
test adaption layer	Added	ETAE keyword
test automation architecture	Added	ETAE keyword
test automation engineer	Added	ETAE keyword
test automation framework	Added	ETAE keyword
test automation manager	Added	ETAE keyword
test automation solution	Added	ETAE keyword
test automation strategy	Added	ETAE keyword
test case result	Added	ETAE keyword
test definition layer	Added	ETAE keyword
test execution layer	Added	ETAE keyword
test generation layer	Added	ETAE keyword
test hook	Added	ETAE keyword
test process improvement	Added	ETAE keyword

Table of Contents

Acknowledgements

This document was produced by the Glossary working group of the International Software Testing Qualifications Board (ISTQB).

At the time the Glossary version 2.4 was completed the Glossary working group had the following members (alphabetic order):

Armin Beer, Armin Born, Mette Bruhn-Pedersen, Josie Crawford, Ernst Dűring, George Fialkovitz, Matthias Hamburg (Vice Chair), Ian Howles, Gábor Kapros, Ozgur Kisir, Gustavo Marquez-Soza, Judy McKay (Chair), Ninna Morin, Avi Ofer, Ana Paiva, Andres Petterson, Juha Pomppu, Meile Posthuma. Lucjan Stapp.

Erik van Veenendaal created the initial version of this Glossary, maintained it and led the Glossary working group from its inception until March 2014. The editors would like to thank him for his pioneering work and major contributions.

Many more people, who are not mentioned here by name, have contributed to former versions of this Glossary. The editors would like to thank them all for their contributions.

This document was formally released by the General Assembly of the ISTQB on July 4th, 2014.

0. Introduction to this Glossary

0.1 Purpose of this Document

The ISTQB Glossary has two main objectives:

- Support the ISTQB syllabi by defining the terms used in the various syllabi
- Support communication within the international testing community and with its stakeholders by providing a standard testing vocabulary

Much time and effort is wasted both within and between industry, commerce, government and professional and academic institutions when ambiguities arise as a result of the inability to differentiate adequately between such terms as 'statement coverage' and 'decision coverage', 'test suite', 'test specification' and 'test plan', and similar terms which form an interface between various sectors of society. Moreover, the professional or technical use of these terms is often at variance, with different meanings attributed to them.

In compiling this glossary, the working group has sought the views and comments of a broad spectrum of opinion in industry, commerce and government bodies and organizations, with the aim of producing an international testing standard that would gain wide acceptance. Total agreement will rarely, if ever, be achieved in compiling a document of this nature. Contributions to this glossary have been received from testing communities from all over the world.

Many software testers have used BS 7925-1, the British Standard *Glossary of Software Testing Terms,* since its original publication in 1998. The standard was initially developed with a bias toward component testing, but, since its publication, many comments and proposals for new definitions have been submitted to both improve and expand the standard to cover a wider range of software testing. The ISTQB testing glossary has incorporated many of these suggested updates.

ISTQB Member Boards can use the ISTQB Glossary to translate into their local language. These boards may adapt the ISTQB Glossary to their particular language needs.

0.2 Scope

This document contains the definitions of testing terms used in the different ISTQB syllabi. It focuses on terms that have a specific meaning in testing. Related non-testing terms are also included if they play a major role in testing, such as terms used in software quality assurance and software lifecycle models. However, most terms of other software engineering disciplines that are used in different ISTQB syllabi are not covered in this document. For instance, the terms commonly used in Agile software development are not included in this document. The Foundation Extension Agile Tester syllabus refers to a number of well-accepted Internet resources that provide appropriate definitions.

0.3 Glossary Structure

Arrangement

The glossary has been arranged in a single section of definitions ordered alphabetically. Some terms are preferred to other synonymous ones, in which case, the definition of the preferred term appears, with the synonymous ones referring to that. For example *structural testing* is a synonym for the preferred term *white box testing*. For synonyms, the "See" indicator is used.

"See also" cross-references are also used. They assist the user to quickly navigate to related terms. "See also" cross-references are constructed for relationships such as broader term to a narrower term, and overlapping meaning between two terms.

Keywords

The ISTQB Glossary contains many terms for different reasons. Some are provided to "just" support the reader of an ISTQB syllabus in understanding the text. Some are there because the term was used in a previous version of a syllabus and the principle of backwards compatibility is being applied. However, probably the most important terms are the (examinable) keywords that are explicitly identified by one or more ISTQB syllabus.

To support testing professionals who are preparing for exams, the keywords for each syllabus are identified. The syllabus is indicated to the left side of the term. The principle of inheritance is applicable, e.g., for an ISTQB Advanced exam one still needs to understand all ISTQB Foundation keywords. The keywords are indicated in the following manner:

F : Keyword ISTQB Foundation syllabus
F-AT : Keyword ISTQB Foundation Extension Agile Tester syllabus
ATM : Keyword ISTQB Advanced – Test Management syllabus
ATA : Keyword ISTQB Advanced – Test Analyst syllabus
ATT : Keyword ISTQB Advanced – Technical Test Analyst syllabus
EITP : Keyword ISTQB Expert – Improving the Testing Process syllabus
ETAE : Keyword ISTQB Expert –Test Automation - Engineering syllabus
ETM : Keyword ISTQB Expert – Test Management syllabus.

Note that if a keyword is identified in a syllabus, but is not the preferred term according to the Glossary, both the keyword and the term it refers to (using the "See" indicator) are labeled with the appropriate syllabus indicator.

References

In this glossary, references are used in two ways:

- Square brackets without the addition of "after", e.g., [ISO 9126], mean that the exact text of the reference is used.
- In case a definition from a reference has been adapted to the context of the ISTQB Glossary by minor changes, the addition "after" is used, e.g., [After ISO 9126].

0.4 Trademarks

In this document the following trademarks are used:

- CMMI and IDEAL are registered trademarks of Carnegie Mellon University
- EFQM is a registered trademark of the EFQM Foundation
- Rational Unified Process (RUP) is a registered trademark of Rational Software Corporation
- STEP is a registered trademark of Software Quality Engineering
- TMap, TPA and TPI Next are registered trademarks of Sogeti Nederland BV
- TMMi is a registered trademark of the TMMi Foundation

1. Definitions

A

abstract test case: See *high level test case*.

acceptance: See *acceptance testing*.

F-AT **acceptance criteria:** The exit criteria that a component or system must satisfy in order to be accepted by a user, customer, or other authorized entity. [IEEE 610]

acceptance testing: Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system. [After IEEE 610]

ATA **accessibility testing:** Testing to determine the ease by which users with disabilities can use a component or system. [Gerrard]

accuracy: The capability of the software product to provide the right or agreed results or effects with the needed degree of precision. [ISO 9126] See also *functionality*.

ATA **accuracy testing:** The process of testing to determine the accuracy of a software product. See also *accuracy.*

EITP **acting (IDEAL):** The phase within the IDEAL model where the improvements are developed, put into practice, and deployed across the organization. The acting phase consists of the activities: create solution, pilot/test solution, refine solution and implement solution. See also *IDEAL*.

action word driven testing: See *keyword-driven testing*

actor: User or any other person or system that interacts with the system under test in a specific way.

actual outcome: See *actual result*.

actual result: The behavior produced/observed when a component or system is tested.

ad hoc review: See *informal review*.

ad hoc testing: Testing carried out informally; no formal test preparation takes place, no recognized test design technique is used, there are no expectations for results and arbitrariness guides the test execution activity.

ATT **adaptability:** The capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered. [ISO 9126] See also *portability*.

F-AT **agile manifesto:** A statement on the values that underpin agile software development. The values are:
- individuals and interactions over processes and tools
- working software over comprehensive documentation
- customer collaboration over contract negotiation
- responding to change over following a plan.

F-AT EITP **agile software development:** A group of software development methodologies based on iterative incremental development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams.

EITP **agile testing:** Testing practice for a project using agile software development methodologies, incorporating techniques and methods, such as extreme programming (XP), treating development as the customer of testing and emphasizing the test-first design paradigm. See also *test-driven development*.

algorithm test: [TMap] See *branch testing*.

F **alpha testing:** Simulated or actual operational testing by potential users/customers or an independent test team at the developers' site, but outside the development organization. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing.

ETM **analytical testing**: Testing based on a systematic analysis of e.g., product risks or requirements.

ATT **analyzability:** The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified. [ISO 9126] See also *maintainability*.

analyzer: See *static analyzer*.

ATM **anomaly:** Any condition that deviates from expectation based on requirements specifications, design documents, user documents, standards, etc. or from someone's perception or experience. Anomalies may be found during, but not limited to, reviewing, testing, analysis, compilation, or use of software products or applicable documentation. [IEEE 1044] See also *bug, defect, deviation, error, fault, failure, incident, problem*.

ATT **anti-pattern**: Repeated action, process, structure or reusable solution that initially appears to be beneficial and is commonly used but is ineffective and/or counterproductive in practice.

API: Acronym for Application Programming Interface.

ETAE **API testing:** Testing performed by submitting commands to the software under test using programming interfaces of the application directly.

arc testing: See *branch testing*.

EITP **assessment report:** A document summarizing the assessment results, e.g. conclusions, recommendations and findings. See also *process assessment.*

EITP **assessor:** A person who conducts an assessment; any member of an assessment team.

ATT **atomic condition:** A condition that cannot be decomposed, i.e., a condition that does not contain two or more single conditions joined by a logical operator (AND, OR, XOR).

F **attack:** Directed and focused attempt to evaluate the quality, especially reliability, of a test object by attempting to force specific failures to occur. See also *negative testing.*

attack-based testing: An experience-based testing technique that uses software attacks to induce failures, particularly security related failures. *See also* attack.

ATA **attractiveness:** The capability of the software product to be attractive to the user. [ISO 9126] See also *usability*.

ATM **audit:** An independent evaluation of software products or processes to ascertain compliance to standards, guidelines, specifications, and/or procedures based on objective criteria, including documents that specify:
(1) the form or content of the products to be produced
(2) the process by which the products shall be produced
(3) how compliance to standards or guidelines shall be measured. [IEEE 1028]

audit trail: A path by which the original input to a process (e.g. data) can be traced back through the process, taking the process output as a starting point. This facilitates defect analysis and allows a process audit to be carried out. [After TMap]

automated testware: Testware used in automated testing, such as tool scripts.

availability: The degree to which a component or system is operational and accessible when required for use. Often expressed as a percentage. [IEEE 610]

B

back-to-back testing: Testing in which two or more variants of a component or system are executed with the same inputs, the outputs compared, and analyzed in cases of discrepancies. [IEEE 610]

EITP **balanced scorecard:** A strategic tool for measuring whether the operational activities of a company are aligned with its objectives in terms of business vision and strategy. See also *corporate dashboard, scorecard.*

baseline: A specification or software product that has been formally reviewed or agreed upon, that thereafter serves as the basis for further development, and that can be changed only through a formal change control process. [After IEEE 610]

basic block: A sequence of one or more consecutive executable statements containing no branches. Note: A node in a control flow graph represents a basic block.

basis test set: A set of test cases derived from the internal structure of a component or specification to ensure that 100% of a specified coverage criterion will be achieved.

bebugging: [Abbott] See *fault seeding*.

behavior: The response of a component or system to a set of input values and preconditions.

benchmark test: (1) A standard against which measurements or comparisons can be made. (2) A test that is be used to compare components or systems to each other or to a standard as in (1). [After IEEE 610]

bespoke software: Software developed specifically for a set of users or customers. The opposite is off-the-shelf software.

best practice: A superior method or innovative practice that contributes to the improved performance of an organization under given context, usually recognized as 'best' by other peer organizations.

F **beta testing:** Operational testing by potential and/or existing users/customers at an external site not otherwise involved with the developers, to determine whether or not a component or system satisfies the user/customer needs and fits within the business processes. Beta testing is often employed as a form of external acceptance testing for off-the-shelf software in order to acquire feedback from the market.

big-bang testing: An integration testing approach in which software elements, hardware elements, or both are combined all at once into a component or an overall system, rather than in stages. [After IEEE 610] See also *integration testing*.

black box technique: See *black box test design technique*.

F ATA **black box test design technique:** Procedure to derive and/or select test cases based on an analysis of the specification, either functional or non-functional, of a component or system without reference to its internal structure.

F **black box testing:** Testing, either functional or non-functional, without reference to the internal structure of the component or system.

blocked test case: A test case that cannot be executed because the preconditions for its execution are not fulfilled.

bottom-up testing: An incremental approach to integration testing where the lowest level components are tested first, and then used to facilitate the testing of higher level components. This process is repeated until the component at the top of the hierarchy is tested. See also *integration testing*.

boundary value: An input value or output value which is on the edge of an equivalence partition or at the smallest incremental distance on either side of an edge, for example the minimum or maximum value of a range.

F ATA **boundary value analysis:** A black box test design technique in which test cases are designed based on boundary values. See also *boundary value.*

boundary value coverage: The percentage of boundary values that have been exercised by a test suite.

boundary value testing: See *boundary value analysis*.

branch: A basic block that can be selected for execution based on a program construct in which one of two or more alternative program paths is available, e.g. case, jump, go to, if-then-else.

branch condition: See *condition*.

branch condition combination coverage: See *multiple condition coverage*.

branch condition combination testing: See *multiple condition testing*.

branch condition coverage: See *condition coverage*.

branch coverage: The percentage of branches that have been exercised by a test suite. 100% branch coverage implies both 100% decision coverage and 100% statement coverage.

branch testing: A white box test design technique in which test cases are designed to execute branches.

buffer: A device or storage area used to store data temporarily for differences in rates of data flow, time or occurrence of events, or amounts of data that can be handled by the devices or processes involved in the transfer or use of the data. [IEEE 610]

buffer overflow: A memory access failure due to the attempt by a process to store data beyond the boundaries of a fixed length buffer, resulting in overwriting of adjacent memory areas or the raising of an overflow exception. See also *buffer.*

F **bug:** See *defect*.

bug report: See *defect report.*

bug taxonomy: *See defect taxonomy.*

bug tracking tool: See *defect management tool.*

F-AT **build verification test:** A set of automated tests which validates the integrity of each new build and verifies its key/core functionality, stability and testability. It is an industry practice when a high frequency of build releases occurs (e.g., agile projects) and it is run on every new build before the build is released for further testing. See also *regression testing, smoke test*.

burndown chart: A publicly displayed chart that depicts the outstanding effort versus time in an iteration. It shows the status and trend of completing the tasks of the iteration. The X-axis typically represents days in the sprint, while the Y-axis is the remaining effort (usually either in ideal engineering hours or story points).

business process-based testing: An approach to testing in which test cases are designed based on descriptions and/or knowledge of business processes.

BVT: See *build verification test*.

C

call graph: An abstract representation of calling relationships between subroutines in a program.

ATM EITP **Capability Maturity Model Integration:** A framework that describes the key elements of an effective product development and maintenance process. The Capability Maturity Model Integration covers best-practices for planning, engineering and managing product development and maintenance. [CMMI]

ETAE **capture/playback:** A test automation approach, where inputs to the test object are recorded during manual testing in order to generate automated test scripts that could be executed later (i.e. replayed).

ATT **capture/playback tool:** A type of test execution tool where inputs are recorded during manual testing in order to generate automated test scripts that can be executed later (i.e. replayed). These tools are often used to support automated regression testing.

capture/replay tool: See *capture/playback tool*.

CASE: Acronym for Computer Aided Software Engineering.

CAST: Acronym for Computer Aided Software Testing. See also *test automation*.

EITP **causal analysis:** The analysis of defects to determine their root cause. [CMMI]

cause-effect analysis: See *cause-effect graphing*.

cause-effect decision table: See *decision table*.

EITP **cause-effect diagram:** A graphical representation used to organize and display the interrelationships of various possible root causes of a problem. Possible causes of a real or potential defect or failure are organized in categories and subcategories in a horizontal tree-structure, with the (potential) defect or failure as the root node. [After Juran]

EITP **cause-effect graph:** A graphical representation of inputs and/or stimuli (causes) with their associated outputs (effects), which can be used to design test cases.

ATA **cause-effect graphing:** A black box test design technique in which test cases are designed from cause-effect graphs. [BS 7925/2]

certification: The process of confirming that a component, system or person complies with its specified requirements, e.g. by passing an exam.

change control: See *configuration control.*

change control board: See *configuration control board*.

EITP **change management:** (1) A structured approach to transitioning individuals, and organizations from a current state to a desired future state. (2) Controlled way to effect a change, or a proposed change, to a product or service. See also *configuration management.*

ATT **changeability:** The capability of the software product to enable specified modifications to be implemented. [ISO 9126] See also *maintainability*.

charter: See *test charter.*

checker: See *reviewer*.

ATA **checklist-based testing:** An experience-based test design technique whereby the experienced tester uses a high-level list of items to be noted, checked, or remembered, or a set of rules or criteria against which a product has to be verified.

Chow's coverage metrics: See *N-switch coverage*. [Chow]

classification tree: A tree showing equivalence partitions hierarchically ordered, which is used to design test cases in the classification tree method. See also *classification tree method.*

ATA **classification tree method:** A black box test design technique in which test cases, described by means of a classification tree, are designed to execute combinations of representatives of input and/or output domains. [Grochtmann] See also *combinatorial testing.*

clear-box testing: See *white-box testing*.

CLI: Acronym for Command-Line Interface.

ETAE **CLI testing:** Testing performed by submitting commands to the software under test using a dedicated command-line interface.

ATM EITP **CMMI:** See *Capability Maturity Model Integration.*

code: Computer instructions and data definitions expressed in a programming language or in a form output by an assembler, compiler or other translator. [IEEE 610]

code analyzer: See *static code analyzer*.

F **code coverage:** An analysis method that determines which parts of the software have been executed (covered) by the test suite and which parts have not been executed, e.g. statement coverage, decision coverage or condition coverage.

code-based testing: See *white box testing*.

EITP **codependent behavior:** Excessive emotional or psychological dependence on another person, specifically in trying to change that person's current (undesirable) behavior while supporting them in continuing that behavior. For example, in software testing, complaining about late delivery to test and yet enjoying the necessary "heroism" working additional hours to make up time when delivery is running late, therefore reinforcing the lateness.

ATT **co-existence:** The capability of the software product to co-exist with other independent software in a common environment sharing common resources. [ISO 9126] See also *portability.*

ATA **combinatorial testing:** A means to identify a suitable subset of test combinations to achieve a predetermined level of coverage when testing an object with multiple parameters and where those parameters themselves each have several values, which gives rise to more combinations than are feasible to test in the time allowed. See also *classification tree method*, *n-wise testing, pairwise testing*, *orthogonal array testing*.

F **Commercial Off-The-Shelf software**: See *off-the-shelf software*.

comparator: See *test comparator*.

compatibility testing: See *interoperability testing*.

F **compiler:** A software tool that translates programs expressed in a high order language into their machine language equivalents. [IEEE 610]

complete testing: See *exhaustive testing*.

completion criteria: See *exit criteria*.

F **complexity:** The degree to which a component or system has a design and/or internal structure that is difficult to understand, maintain and verify. See also *cyclomatic complexity.*

compliance: The capability of the software product to adhere to standards, conventions or regulations in laws and similar prescriptions. [ISO 9126]

compliance testing: The process of testing to determine the compliance of the component or system.

component: A minimal software item that can be tested in isolation.

component integration testing: Testing performed to expose defects in the interfaces and interaction between integrated components.

component specification: A description of a component's function in terms of its output values for specified input values under specified conditions, and required non-functional behavior (e.g. resource-utilization).

F **component testing:** The testing of individual software components. [After IEEE 610]

compound condition: Two or more single conditions joined by means of a logical operator (AND, OR or XOR), e.g. 'A>B AND C>1000'.

ATA **concrete test case:** See *low level test case*.

concurrency testing: Testing to determine how the occurrence of two or more activities within the same interval of time, achieved either by interleaving the activities or by simultaneous execution, is handled by the component or system. [After IEEE 610]

condition: A logical expression that can be evaluated as True or False, e.g. A>B. See also *condition testing.*

condition combination coverage: See *multiple condition coverage*.

condition combination testing: See *multiple condition testing*.

condition coverage: The percentage of condition outcomes that have been exercised by a test suite. 100% condition coverage requires each single condition in every decision statement to be tested as True and False.

condition determination coverage: See *modified condition decision coverage*.

condition determination testing: See *modified condition decision testing.*

condition outcome: The evaluation of a condition to True or False.

ATT **condition testing:** A white box test design technique in which test cases are designed to execute condition outcomes.

ETM **confidence interval**: In managing project risks, the period of time within which a contingency action must be implemented in order to be effective in reducing the impact of the risk.

confidence test: See *smoke test*.

configuration: The composition of a component or system as defined by the number, nature, and interconnections of its constituent parts.

configuration auditing: The function to check on the contents of libraries of configuration items, e.g. for standards compliance. [IEEE 610]

configuration control: An element of configuration management, consisting of the evaluation, co-ordination, approval or disapproval, and implementation of changes to configuration items after formal establishment of their configuration identification. [IEEE 610]

configuration control board (CCB): A group of people responsible for evaluating and approving or disapproving proposed changes to configuration items, and for ensuring implementation of approved changes. [IEEE 610]

configuration identification: An element of configuration management, consisting of selecting the configuration items for a system and recording their functional and physical characteristics in technical documentation. [IEEE 610]

F-AT **configuration item:** An aggregation of hardware, software or both, that is designated for configuration management and treated as a single entity in the configuration management process. [IEEE 610]

F F-AT **configuration management:** A discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements. [IEEE 610]

F **configuration management tool:** A tool that provides support for the identification and control of configuration items, their status over changes and versions, and the release of baselines consisting of configuration items.

configuration testing: See *portability testing*.

F **confirmation testing:** Testing that runs test cases that failed the last time they were run, in order to verify the success of corrective actions.

conformance testing: See *compliance testing*.

consistency: The degree of uniformity, standardization, and freedom from contradiction among the documents or parts of a component or system. [IEEE 610]

ETM **consultative testing**: Testing driven by the advice and guidance of appropriate experts from outside the test team (e.g., technology experts and/or business domain experts).

EITP **content-based model:** A process model providing a detailed description of good engineering practices, e.g. test practices.

content reference model: See *content-based model.*

EITP **continuous representation:** A capability maturity model structure wherein capability levels provide a recommended order for approaching process improvement within specified process areas. [CMMI]

ETM **control chart**: A statistical process control tool used to monitor a process and determine whether it is statistically controlled. It graphically depicts the average value and the upper and lower control limits (the highest and lowest values) of a process.

F **control flow:** A sequence of events (paths) in the execution through a component or system.

ATT **control flow analysis:** A form of static analysis based on a representation of unique paths (sequences of events) in the execution through a component or system. Control flow analysis evaluates the integrity of control flow structures, looking for possible control flow anomalies such as closed loops or logically unreachable process steps.

control flow graph: An abstract representation of all possible sequences of events (paths) in the execution through a component or system.

control flow path: See *path*.

ATT **control flow testing**: An approach to structure-based testing in which test cases are designed to execute specific sequences of events. Various techniques exist for control flow testing, e.g., decision testing, condition testing, and path testing, that each have their specific approach and level of control flow coverage. *See also* decision testing, condition testing, path testing.

ETM **convergence metric:** A metric that shows progress toward a defined criterion, e.g., convergence of the total number of test executed to the total number of tests planned for execution.

conversion testing: Testing of software used to convert data from existing systems for use in replacement systems.

EITP **corporate dashboard:** A dashboard-style representation of the status of corporate performance data. See also *balanced scorecard, dashboard.*

cost of quality: The total costs incurred on quality activities and issues and often split into prevention costs, appraisal costs, internal failure costs and external failure costs.

F **COTS:** Acronym for Commercial Off-The-Shelf software. See *off-the-shelf software.*

ETAE **coverage:** The degree, expressed as a percentage, to which a specified coverage item has been exercised by a test suite.

coverage analysis: Measurement of achieved coverage to a specified coverage item during test execution referring to predetermined criteria to determine whether additional testing is required and if so, which test cases are needed.

coverage item: An entity or property used as a basis for test coverage, e.g. equivalence partitions or code statements.

coverage measurement tool: See *coverage tool.*

F **coverage tool:** A tool that provides objective measures of what structural elements, e.g. statements, branches have been exercised by a test suite.

EITP **critical success factor:** An element necessary for an organization or project to achieve its mission. Critical success factors are the critical factors or activities required for ensuring the success.

ATM EITP **Critical Testing Processes:** A content-based model for test process improvement built around twelve critical processes. These include highly visible processes, by which peers and management judge competence and mission-critical processes in which performance affects the company's profits and reputation. See also *content-based model.*

ATM **CTP:** See *Critical Testing Processes.*

EITP

custom software: See *bespoke software*.

ATM **custom tool:** A software tool developed specifically for a set of users or customers.

ATT **cyclomatic complexity:** The maximum number of linear, independent paths through a program. Cyclomatic complexity may be computed as: L – N + 2P, where
- L = the number of edges/links in a graph
- N = the number of nodes in a graph
- P = the number of disconnected parts of the graph (e.g. a called graph or subroutine) [After McCabe]

cyclomatic number: See *cyclomatic complexity*.

D

daily build: A development activity whereby a complete system is compiled and linked every day (often overnight), so that a consistent system is available at any time including all latest changes.

ETM **dashboard:** A representation of dynamic measurements of operational performance for some organization or activity, using metrics represented via metaphors such as visual 'dials', 'counters', and other devices resembling those on the dashboard of an automobile, so that the effects of events or activities can be easily understood and related to operational goals. See also *corporate dashboard, scorecard.*

data definition: An executable statement where a variable is assigned a value.

F ATT ETAE **data-driven testing:** A scripting technique that stores test input and expected results in a table or spreadsheet, so that a single control script can execute all of the tests in the table. Data-driven testing is often used to support the application of test execution tools such as capture/playback tools. [Fewster and Graham] See also *keyword-driven testing*.

F **data flow:** An abstract representation of the sequence and possible changes of the state of data objects, where the state of an object is any of: creation, usage, or destruction. [Beizer]

ATT **data flow analysis:** A form of static analysis based on the definition and usage of variables.

data flow coverage: The percentage of definition-use pairs that have been exercised by a test suite.

data flow testing: A white box test design technique in which test cases are designed to execute definition-use pairs of variables.

data integrity testing: See *database integrity testing*.

data quality; An attribute of data that indicates correctness with respect to some pre-defined criteria, e.g., business expectations, requirements on data integrity, data consistency.

database integrity testing: Testing the methods and processes used to access and manage the data(base), to ensure access methods, processes and data rules function as expected and that during access to the database, data is not corrupted or unexpectedly deleted, updated or created.

dd-path: A path between two decisions of an algorithm, or two decision nodes of a corresponding graph, that includes no other decisions. See also *path.*

dead code: See *unreachable code*.

debugger: See *debugging tool*.

F **debugging:** The process of finding, analyzing and removing the causes of failures in software.

F ATT **debugging tool:** A tool used by programmers to reproduce failures, investigate the state of programs and find the corresponding defect. Debuggers enable programmers to execute programs step by step, to halt a program at any program statement and to set and examine program variables.

decision: A program point at which the control flow has two or more alternative routes. A node with two or more links to separate branches.

decision condition coverage: The percentage of all condition outcomes and decision outcomes that have been exercised by a test suite. 100% decision condition coverage implies both 100% condition coverage and 100% decision coverage.

ATT **decision condition testing:** A white box test design technique in which test cases are designed to execute condition outcomes and decision outcomes.

F **decision coverage:** The percentage of decision outcomes that have been exercised by a test suite. 100% decision coverage implies both 100% branch coverage and 100% statement coverage.

decision outcome: The result of a decision (which therefore determines the branches to be taken).

decision table: A table showing combinations of inputs and/or stimuli (causes) with their associated outputs and/or actions (effects), which can be used to design test cases.

F ATA **decision table testing:** A black box test design technique in which test cases are designed to execute the combinations of inputs and/or stimuli (causes) shown in a decision table. [Veenendaal04] See also *decision table.*

decision testing: A white box test design technique in which test cases are designed to execute decision outcomes.

F ATM **defect:** A flaw in a component or system that can cause the component or system to fail to perform its required function, e.g. an incorrect statement or data definition. A defect, if encountered during execution, may cause a failure of the component or system.

ATA defect-based technique: See *defect-based test design technique*.

ATA **defect-based test design technique:** A procedure to derive and/or select test cases targeted at one or more defect types, with tests being developed from what is known about the specific defect type. See also *defect taxonomy.*

defect category: See *defect type.*

F **defect density:** The number of defects identified in a component or system divided by the size of the component or system (expressed in standard measurement terms, e.g. lines-of-code, number of classes or function points).

EITP **Defect Detection Percentage (DDP):** The number of defects found by a test level, divided by the number found by that test level and any other means afterwards. See also *escaped defects.*

defect management: The process of recognizing, investigating, taking action and disposing of defects. It involves recording defects, classifying them and identifying the impact. [After IEEE 1044]

ATM **defect management committee:** A cross-functional team of stakeholders who manage reported defects from initial detection to ultimate resolution (defect removal, defect deferral, or report cancellation). In some cases, the same team as the configuration control board. *See also* configuration control board.

defect management tool: A tool that facilitates the recording and status tracking of defects and changes. They often have workflow-oriented facilities to track and control the allocation, correction and re-testing of defects and provide reporting facilities. See also *incident management tool*.

defect masking: An occurrence in which one defect prevents the detection of another. [After IEEE 610]

defect report: A document reporting on any flaw in a component or system that can cause the component or system to fail to perform its required function. [After IEEE 829]

F-AT ATA **defect taxonomy:** A system of (hierarchical) categories designed to be a useful aid for reproducibly classifying defects.

defect tracking tool: See *defect management tool*.

ATM **defect triage committee**: *See* defect management committee.

defect type: An element in a taxonomy of defects. Defect taxonomies can be identified with respect to a variety of considerations, including, but not limited to:

- Phase or development activity in which the defect is created, e.g., a specification error or a coding error
- Characterization of defects, e.g., an "off-by-one" defect
- Incorrectness, e.g., an incorrect relational operator, a programming language syntax error, or an invalid assumption
- Performance issues, e.g., excessive execution time, insufficient availability.

ATT **definition-use pair:** The association of a definition of a variable with the subsequent use of that variable. Variable uses include computational (e.g. multiplication) or to direct the execution of a path ("predicate" use).

deliverable: Any (work) product that must be delivered to someone other than the (work) product's author.

EITP **Deming cycle:** An iterative four-step problem-solving process, (plan-do-check-act), typically used in process improvement. [After Deming]

design-based testing: An approach to testing in which test cases are designed based on the architecture and/or detailed design of a component or system (e.g. tests of interfaces between components or systems).

desk checking: Testing of software or a specification by manual simulation of its execution. See also *static testing.*

development testing: Formal or informal testing conducted during the implementation of a component or system, usually in the development environment by developers. [After IEEE 610]

deviation: See *incident*.

deviation report: See *incident report*.

EITP **diagnosing (IDEAL):** The phase within the IDEAL model where it is determined where one is, relative to where one wants to be. The diagnosing phase consists of the activities: characterize current and desired states and develop recommendations. See also *IDEAL*.

dirty testing: See *negative testing*.

documentation testing: Testing the quality of the documentation, e.g. user guide or installation guide.

domain: The set from which valid input and/or output values can be selected.

ATA **domain analysis:** A black box test design technique that is used to identify efficient and effective test cases when multiple variables can or should be tested together. It builds on and generalizes equivalence partitioning and boundary values analysis. *See also* boundary value analysis, equivalence partitioning.

F ETAE **driver:** A software component or test tool that replaces a component that takes care of the control and/or the calling of a component or system. [After TMap]

ATT **dynamic analysis:** The process of evaluating behavior, e.g. memory performance, CPU usage, of a system or component during execution. [After IEEE 610]

F **dynamic analysis tool:** A tool that provides run-time information on the state of the software code. These tools are most commonly used to identify unassigned pointers, check pointer arithmetic and to monitor the allocation, use and de-allocation of memory and to flag memory leaks.

dynamic comparison: Comparison of actual and expected results, performed while the software is being executed, for example by a test execution tool.

F **dynamic testing:** Testing that involves the execution of the software of a component or **system.**

E

ATM **effectiveness**: The capability of producing an intended result. See also *efficiency*.

ATM ATT **efficiency:** (1) The capability of the software product to provide appropriate performance, relative to the amount of resources used under stated conditions. [ISO 9126]
(2) The capability of a process to produce the intended outcome, relative to the amount of resources used

efficiency testing: The process of testing to determine the efficiency of a software product.

EITP **EFQM (European Foundation for Quality Management) excellence model:** A non-prescriptive framework for an organisation's quality management system, defined and owned by the European Foundation for Quality Management, based on five 'Enabling' criteria (covering what an organisation does), and four 'Results' criteria (covering what an organisation achieves).

elementary comparison testing: A black box test design technique in which test cases are designed to execute combinations of inputs using the concept of modified condition decision coverage. [TMap]

embedded iterative development model: A development lifecycle sub-model that applies an iterative approach to detailed design, coding and testing within an overall sequential model. In this case, the high level design documents are prepared and approved for the entire project but the actual detailed design, code development and testing are conducted in iterations.

EITP **emotional intelligence:** The ability, capacity, and skill to identify, assess, and manage the emotions of one's self, of others, and of groups.

EMTE: Acronym for Equivalent Manual Test Effort.

emulator: A device, computer program, or system that accepts the same inputs and produces the same outputs as a given system. [IEEE 610] See also *simulator*.

F **entry criteria:** The set of generic and specific conditions for permitting a process to go forward with a defined task, e.g. test phase. The purpose of entry criteria is to prevent a task from starting which would entail more (wasted) effort compared to the effort needed to remove the failed entry criteria. [Gilb and Graham]

entry point: An executable statement or process step which defines a point at which a given process is intended to begin.

equivalence class: See *equivalence partition*.

equivalence partition: A portion of an input or output domain for which the behavior of a component or system is assumed to be the same, based on the specification.

equivalence partition coverage: The percentage of equivalence partitions that have been exercised by a test suite.

F ATA **equivalence partitioning:** A black box test design technique in which test cases are designed to execute representatives from equivalence partitions. In principle test cases are designed to cover each partition at least once.

ETAE **equivalent manual test effort:** Effort required for running tests manually.

F **error:** A human action that produces an incorrect result. [After IEEE 610]

F ATA **error guessing:** A test design technique where the experience of the tester is used to anticipate what defects might be present in the component or system under test as a result of errors made, and to design tests specifically to expose them.

error seeding: See *fault seeding.*

error seeding tool: See *fault seeding tool.*

error tolerance: The ability of a system or component to continue normal operation despite the presence of erroneous inputs. [After IEEE 610].

escaped defect: A defect that was not detected in a previous test level which is supposed to find such type of defects. See also *Defect Detection Percentage.*

EITP **establishing (IDEAL):** The phase within the IDEAL model where the specifics of how an organization will reach its destination are planned. The establishing phase consists of the activities: set priorities, develop approach and plan actions. See also *IDEAL*.

evaluation: See *testing*.

exception handling: Behavior of a component or system in response to erroneous input, from either a human user or from another component or system, or to an internal failure.

executable statement: A statement which, when compiled, is translated into object code, and which will be executed procedurally when the program is running and may perform an action on data.

exercised: A program element is said to be exercised by a test case when the input value causes the execution of that element, such as a statement, decision, or other structural element.

F **exhaustive testing:** A test approach in which the test suite comprises all combinations of input values and preconditions.

F ATM ATA **exit criteria:** The set of generic and specific conditions, agreed upon with the stakeholders for permitting a process to be officially completed. The purpose of exit criteria is to prevent a task from being considered completed when there are still outstanding parts of the task which have not been finished. Exit criteria are used to report against and to plan when to stop testing. [After Gilb and Graham]

exit point: An executable statement or process step which defines a point at which a given process is intended to cease..

expected outcome: See *expected result*.

expected result: The behavior predicted by the specification, or another source, of the component or system under specified conditions.

ATA **experience-based technique:** See *experience-based test design technique.*

F ATA **experience-based test design technique:** Procedure to derive and/or select test cases based on the tester's experience, knowledge and intuition.

experience-based testing: Testing based on the tester's experience, knowledge and intuition.

F F-AT ATA **exploratory testing:** An informal test design technique where the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests. [After Bach]

EITP **extreme programming (XP):** A software engineering methodology used within agile software development whereby core practices are programming in pairs, doing extensive code review, unit testing of all code, and simplicity and clarity in code. See also *agile software development.*

F

factory acceptance testing: Acceptance testing conducted at the site at which the product is developed and performed by employees of the supplier organization, to determine whether or not a component or system satisfies the requirements, normally including hardware as well as software. See also *alpha testing.*

fail: A test is deemed to fail if its actual result does not match its expected result.

failover testing: Testing by simulating failure modes or actually causing failures in a controlled environment. Following a failure, the failover mechanism is tested to ensure that data is not lost or corrupted and that any agreed service levels are maintained (e.g., function availability or response times). See also *recoverability testing*.

F **failure:** Deviation of the component or system from its expected delivery, service or result.
ATM [After Fenton]

failure mode: The physical or functional manifestation of a failure. For example, a system in failure mode may be characterized by slow operation, incorrect outputs, or complete termination of execution. [IEEE 610]

EITP **Failure Mode and Effect Analysis (FMEA):** A systematic approach to risk identification and analysis of identifying possible modes of failure and attempting to prevent their occurrence. See also *Failure Mode, Effect and Criticality Analysis (FMECA).*

Failure Mode, Effects, and Criticality Analysis (FMECA): An extension of FMEA, as in addition to the basic FMEA, it includes a criticality analysis, which is used to chart the probability of failure modes against the severity of their consequences. The result highlights failure modes with relatively high probability and severity of consequences, allowing remedial effort to be directed where it will produce the greatest value. See also *Failure Mode and Effect Analysis (FMEA).*

F **failure rate:** The ratio of the number of failures of a given category to a given unit of measure, e.g. failures per unit of time, failures per number of transactions, failures per number of computer runs. [IEEE 610]

ATM **false-fail result**: A test result in which a defect is reported although no such defect actually exists in the test object.

ATM **false-negative result**: See *false-pass result*.

ATM **false-pass result**: A test result which fails to identify the presence of a defect that is actually present in the test object.

ATM **false-positive result**: See *false-fail result*.

F **fault:** See *defect*.

F **fault attack:** See *attack*.

fault density: See *defect density*.

Fault Detection Percentage (FDP): See *Defect Detection Percentage (DDP).*

fault injection: The process of intentionally adding defects to a system for the purpose of finding out whether the system can detect, and possibly recover from, a defect. Fault injection intended to mimic failures that might occur in the field. See also *fault tolerance*.

fault masking: See *defect masking*.

fault seeding: The process of intentionally adding defects to those already in the component or system for the purpose of monitoring the rate of detection and removal, and estimating the number of remaining defects. Fault seeding is typically part of development (pre-release) testing and can be performed at any test level (component, integration, or system). [After IEEE 610]

ATT **fault seeding tool:** A tool for seeding (i.e. intentionally inserting) faults in a component or system.

fault tolerance: The capability of the software product to maintain a specified level of performance in cases of software faults (defects) or of infringement of its specified interface. [ISO 9126] See also *reliability, robustness*.

EITP **Fault Tree Analysis (FTA):** A technique used to analyze the causes of faults (defects). The technique visually models how logical relationships between failures, human errors, and external events can combine to cause specific faults to disclose.

feasible path: A path for which a set of input values and preconditions exists which causes it to be executed.

feature: An attribute of a component or system specified or implied by requirements documentation (for example reliability, usability or design constraints). [After IEEE 1008]

ETM **feature-driven development**: An iterative and incremental software development process driven from a client-valued functionality (feature) perspective. Feature-driven development is mostly used in agile software development. See also *agile software development.*

F **field testing:** See *beta testing*.

finite state machine: A computational model consisting of a finite number of states and transitions between those states, possibly with accompanying actions. [IEEE 610]

finite state testing: See *state transition testing*.

fishbone diagram: See *cause-effect diagram.*

F **formal review:** A review characterized by documented procedures and requirements, e.g. inspection.

frozen test basis: A test basis document that can only be amended by a formal change control process. See also *baseline*.

Function Point Analysis (FPA): Method aiming to measure the size of the functionality of an information system. The measurement is independent of the technology. This measurement may be used as a basis for the measurement of productivity, the estimation of the needed resources, and project control.

functional integration: An integration approach that combines the components or systems for the purpose of getting a basic functionality working early. See also *integration testing*.

F **functional requirement:** A requirement that specifies a function that a component or system must perform. [IEEE 610]

functional test design technique: Procedure to derive and/or select test cases based on an analysis of the specification of the functionality of a component or system without reference to its internal structure. See also *black box test design technique*.

F **functional testing:** Testing based on an analysis of the specification of the functionality of a component or system. See also *black box testing*.

functionality: The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions. [ISO 9126]

functionality testing: The process of testing to determine the functionality of a software product.

G

ETAE **generic test automation architecture**: Representation of the layers, components, and interfaces of a test automation architecture, allowing for a structured and modular approach to implement test automation.

glass box testing: See *white box testing*.

EITP **Goal Question Metric:** An approach to software measurement using a three-level model: conceptual level (goal), operational level (question) and quantitative level (metric).

EITP **GQM:** See Goal Question Metric.

GUI: Acronym for Graphical User Interface.

ETAE **GUI testing**: Testing performed by interacting with the software under test via the graphical user interface.

H

hardware-software integration testing: Testing performed to expose defects in the interfaces and interaction between hardware and software components. See also *integration testing*.

hazard analysis: A technique used to characterize the elements of risk. The result of a hazard analysis will drive the methods used for development and testing of a system. See also *risk analysis.*

ATA **heuristic evaluation:** A usability review technique that targets usability problems in the user interface or user interface design. With this technique, the reviewers examine the interface and judge its compliance with recognized usability principles (the "heuristics").

ATA **high level test case:** A test case without concrete (implementation level) values for input data and expected results. Logical operators are used; instances of the actual values are not yet defined and/or available. See also *low level test case*.

horizontal traceability: The tracing of requirements for a test level through the layers of test documentation (e.g. test plan, test design specification, test case specification and test procedure specification or test script).

hyperlink: A pointer within a web page that leads to other web pages.

ATT **hyperlink test tool**: A tool used to check that no broken hyperlinks are present on a web site.

I

EITP **IDEAL:** An organizational improvement model that serves as a roadmap for initiating, planning, and implementing improvement actions. The IDEAL model is named for the five phases it describes: initiating, diagnosing, establishing, acting, and learning.

F **impact analysis:** The assessment of change to the layers of development documentation, test documentation and components, in order to implement a given change to specified requirements.

F **incident:** Any event occurring that requires investigation. [After IEEE 1008]

F **incident logging:** Recording the details of any incident that occurred, e.g. during testing.

F **incident management:** The process of recognizing, investigating, taking action and disposing of incidents. It involves logging incidents, classifying them and identifying the impact. [After IEEE 1044]

F **incident management tool:** A tool that facilitates the recording and status tracking of incidents. They often have workflow-oriented facilities to track and control the allocation, correction and re-testing of incidents and provide reporting facilities. See also *defect management tool.*

F **incident report:** A document reporting on any event that occurred, e.g. during the testing, which requires investigation. [After IEEE 829]

F F-AT **incremental development model:** A development lifecycle where a project is broken into a series of increments, each of which delivers a portion of the functionality in the overall project requirements. The requirements are prioritized and delivered in priority order in the appropriate increment. In some (but not all) versions of this lifecycle model, each subproject follows a 'mini V-model' with its own design, coding and testing phases.

incremental testing: Testing where components or systems are integrated and tested one or some at a time, until all the components or systems are integrated and tested.

F ATM **independence of testing:** Separation of responsibilities, which encourages the accomplishment of objective testing. [After DO-178b]

EITP **indicator:** A measure that can be used to estimate or predict another measure. [ISO 14598]

infeasible path: A path that cannot be exercised by any set of possible input values.

F ATM **informal review:** A review not based on a formal (documented) procedure.

EITP **initiating (IDEAL):** The phase within the IDEAL model where the groundwork is laid for a successful improvement effort. The initiating phase consists of the activities: set context, build sponsorship and charter infrastructure. See also *IDEAL*.

input: A variable (whether stored within a component or outside) that is read by a component.

input domain: The set from which valid input values can be selected. See also *domain*.

input value: An instance of an input. See also *input*.

insourced testing: Testing performed by people who are co-located with the project team but are not fellow employees.

F ATM EITP **inspection:** A type of peer review that relies on visual examination of documents to detect defects, e.g. violations of development standards and non-conformance to higher level documentation. The most formal review technique and therefore always based on a documented procedure. [After IEEE 610, IEEE 1028] See also *peer review*.

inspection leader: See *moderator*.

inspector: See *reviewer*.

ATT **installability:** The capability of the software product to be installed in a specified environment [ISO 9126]. See also *portability*.

installability testing: The process of testing the installability of a software product. See also *portability testing*.

installation guide: Supplied instructions on any suitable media, which guides the installer through the installation process. This may be a manual guide, step-by-step procedure, installation wizard, or any other similar process description.

installation wizard: Supplied software on any suitable media, which leads the installer through the installation process. It normally runs the installation process, provides feedback on installation results, and prompts for options.

instrumentation: The insertion of additional code into the program in order to collect information about program behavior during execution, e.g. for measuring code coverage.

instrumenter: A software tool used to carry out instrumentation.

intake test: A special instance of a smoke test to decide if the component or system is ready for detailed and further testing. An intake test is typically carried out at the start of the test execution phase. See also *smoke test*.

F **integration:** The process of combining components or systems into larger assemblies.

F **integration testing:** Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems. See also *component integration testing, system integration testing*.

integration testing in the large: See *system integration testing*.

integration testing in the small: See *component integration testing*.

interface testing: An integration test type that is concerned with testing the interfaces between components or systems.

interoperability: The capability of the software product to interact with one or more specified components or systems. [After ISO 9126] See also *functionality*.

F ATA **interoperability testing:** The process of testing to determine the interoperability of a software product. See also *functionality testing*.

invalid testing: Testing using input values that should be rejected by the component or system. See also *error tolerance, negative testing*.

ETM **Ishikawa diagram:** See *cause-effect diagram.*

isolation testing: Testing of individual components in isolation from surrounding components, with surrounding components being simulated by stubs and drivers, if needed.

item transmittal report: See *release note*.

F F-AT **iterative development model:** A development lifecycle where a project is broken into a usually large number of iterations. An iteration is a complete development loop resulting in a release (internal or external) of an executable product, a subset of the final product under development, which grows from iteration to iteration to become the final product.

K

key performance indicator: See *performance indicator*.

F ATA ATT ETAE **keyword-driven testing:** A scripting technique that uses data files to contain not only test data and expected results, but also keywords related to the application being tested. The keywords are interpreted by special supporting scripts that are called by the control script for the test. See also *data-driven testing*.

L

LCSAJ: A Linear Code Sequence And Jump, consists of the following three items (conventionally identified by line numbers in a source code listing): the start of the linear sequence of executable statements, the end of the linear sequence, and the target line to which control flow is transferred at the end of the linear sequence.

LCSAJ coverage: The percentage of LCSAJs of a component that have been exercised by a test suite. 100% LCSAJ coverage implies 100% decision coverage.

LCSAJ testing: A white box test design technique in which test cases are designed to execute LCSAJs.

EITP **lead assessor:** The person who leads an assessment. In some cases, for instance CMMi and TMMi when formal assessments are conducted, the lead assessor must be accredited and formally trained.

ATA **learnability:** The capability of the software product to enable the user to learn its application. [ISO 9126] See also *usability*.

EITP **learning (IDEAL):** The phase within the IDEAL model where one learns from experiences and improves one's ability to adopt new processes and technologies in the future. The learning phase consists of the activities: analyze and validate, and propose future actions. See also *IDEAL*.

ETAE **level of intrusion**: The level to which a test object is modified by adjusting it for testability.

ATM **level test plan**: A test plan that typically addresses one test level. See also *test plan.*

EITP **lifecycle model:** A partitioning of the life of a product or project into phases. [CMMI] See also *software lifecycle.*

ETAE **linear scripting:** A simple scripting technique without any control structure in the test scripts.

link testing: See *component integration testing*.

load profile: A specification of the activity which a component or system being tested may experience in production. A load profile consists of a designated number of virtual users who process a defined set of transactions in a specified time period and according to a predefined operational profile. See also *operational profile.*

F **load testing:** A type of performance testing conducted to evaluate the behavior of a component or system with increasing load, e.g. numbers of parallel users and/or numbers of transactions, to determine what load can be handled by the component or system. See also *performance testing, stress testing*.

F **load testing tool:** A tool to support load testing whereby it can simulate increasing load, e.g., numbers of concurrent users and/or transactions within a specified time-period. See also *performance testing tool.*

logic-coverage testing: See *white box testing*. [Myers]

logic-driven testing: See *white box testing*.

ATA **logical test case:** See *high level test case*.

ATA **low level test case:** A test case with concrete (implementation level) values for input data and expected results. Logical operators from high level test cases are replaced by actual values that correspond to the objectives of the logical operators. See also *high level test case*.

M

maintainability: The ease with which a software product can be modified to correct defects, modified to meet new requirements, modified to make future maintenance easier, or adapted to a changed environment. [ISO 9126]

F F-AT ATT **maintainability testing:** The process of testing to determine the maintainability of a software product.

ETAE **maintenance:** Modification of a software product after delivery to correct defects, to improve performance or other attributes, or to adapt the product to a modified environment. [IEEE 1219]

F **maintenance testing:** Testing the changes to an operational system or the impact of a changed environment to an operational system.

man in the middle attack: The interception, mimicking and/or altering and subsequent relaying of communications (e.g., credit card transactions) by a third party such that a user remains unaware of that third party's presence.

ATM **management review:** A systematic evaluation of software acquisition, supply, development, operation, or maintenance process, performed by or on behalf of management that monitors progress, determines the status of plans and schedules, confirms requirements and their system allocation, or evaluates the effectiveness of management approaches to achieve fitness for purpose. [After IEEE 610, IEEE 1028]

EITP **manufacturing-based quality:** A view of quality, whereby quality is measured by the degree to which a product or service conforms to its intended design and requirements. Quality arises from the process(es) used. [After Garvin] See also *product-based quality, transcendent-based quality, user-based quality, value-based quality.*

ATM **master test plan:** A test plan that typically addresses multiple test levels. See also *test plan*.

ATT **maturity:** (1) The capability of an organization with respect to the effectiveness and efficiency of its processes and work practices. See also *Capability Maturity Model Integration*, *Test Maturity Model integration*.
(2) The capability of the software product to avoid failure as a result of defects in the software. [ISO 9126] See also *reliability*.

EITP **maturity level:** Degree of process improvement across a predefined set of process areas in which all goals in the set are attained. [TMMi]

maturity model: A structured collection of elements that describe certain aspects of maturity in an organization, and aid in the definition and understanding of an organization's processes. A maturity model often provides a common language, shared vision and framework for prioritizing improvement actions.

MCDC: See *modified condition decision coverage*.

Mean Time Between Failures: The arithmetic mean (average) time between failures of a system. The MTBF is typically part of a reliability growth model that assumes the failed system is immediately repaired, as a part of a defect fixing process. See also *reliability growth model.*

Mean Time To Repair: The arithmetic mean (average) time a system will take to recover from any failure. This typically includes testing to insure that the defect has been resolved.

EITP **measure:** The number or category assigned to an attribute of an entity by making a measurement. [ISO 14598]

measurement: The process of assigning a number or category to an entity to describe an attribute of that entity. [ISO 14598]

measurement scale: A scale that constrains the type of data analysis that can be performed on it. [ISO 14598]

ATT **memory leak:** A memory access failure due to a defect in a program's dynamic store allocation logic that causes it to fail to release memory after it has finished using it, eventually causing the program and/or other concurrent processes to fail due to lack of memory.

ETM **methodical testing**: Testing based on a standard set of tests, e.g., a checklist, a quality standard, or a set of generalized test cases.

F EITP ETAE **metric:** A measurement scale and the method used for measurement. [ISO 14598]

migration testing: See *conversion testing*.

milestone: A point in time in a project at which defined (intermediate) deliverables and results should be ready.

EITP **mind map:** A diagram used to represent words, ideas, tasks, or other items linked to and arranged around a central keyword or idea. Mind maps are used to generate, visualize, structure, and classify ideas, and as an aid in study, organization, problem solving, decision making, and writing.

F **mistake:** See *error*.

ETM ETAE **model-based testing**: Testing based on a model of the component or system under test, e.g., reliability growth models, usage models such as operational profiles or behavioral models such as decision table or state transition diagram.

F **modeling tool:** A tool that supports the creation, amendment and verification of models of the software or system [Graham].

F ATM **moderator:** The leader and main person responsible for an inspection or other review process.

modified condition decision coverage: The percentage of all single condition outcomes that independently affect a decision outcome that have been exercised by a test case suite. 100% modified condition decision coverage implies 100% decision condition coverage.

modified condition decision testing: A white box test design technique in which test cases are designed to execute single condition outcomes that independently affect a decision outcome.

modified multiple condition coverage: See *modified condition decision coverage*.

modified multiple condition testing: See *modified condition decision testing*.

module: See *component*.

module testing: See *component testing*.

monitor: A software tool or hardware device that runs concurrently with the component or system under test and supervises, records and/or analyses the behavior of the component or system. [After IEEE 610]

F **monitoring tool:** See *monitor.*

monkey testing: Testing by means of a random selection from a large range of inputs and by randomly pushing buttons, ignorant of how the product is being used.

MTBF: See *Mean Time Between Failures.*

MTTR: See *Mean Time To Repair.*

multiple condition: See *compound condition*.

multiple condition coverage: The percentage of combinations of all single condition outcomes within one statement that have been exercised by a test suite. 100% multiple condition coverage implies 100% modified condition decision coverage.

ATT **multiple condition testing:** A white box test design technique in which test cases are designed to execute combinations of single condition outcomes (within one statement).

mutation analysis: A method to determine test suite thoroughness by measuring the extent to which a test suite can discriminate the program from slight variants (mutants) of the program.

mutation testing: See *back-to-back testing*.

ETM **Myers-Briggs Type Indicator (MBTI)**: An indicator of psychological preference representing the different personalities and communication styles of people.

N

N-switch coverage: The percentage of sequences of N+1 transitions that have been exercised by a test suite. [Chow]

N-switch testing: A form of state transition testing in which test cases are designed to execute all valid sequences of N+1 transitions. [Chow] See also *state transition testing*.

n-wise testing: A black box test design technique in which test cases are designed to execute all possible discrete combinations of any set of n input parameters. See also *combinatorial testing, orthogonal array testing, pairwise testing.*

negative testing: Tests aimed at showing that a component or system does not work. Negative testing is related to the testers' attitude rather than a specific test approach or test design technique, e.g. testing with invalid input values or exceptions. [After Beizer].

ATT **neighborhood integration testing:** A form of integration testing where all of the nodes that connect to a given node are the basis for the integration testing.

non-conformity: Non fulfillment of a specified requirement. [ISO 9000]

F **non-functional requirement:** A requirement that does not relate to functionality, but to attributes such as reliability, efficiency, usability, maintainability and portability.

non-functional test design technique: Procedure to derive and/or select test cases for non-functional testing based on an analysis of the specification of a component or system without reference to its internal structure. See also *black box test design technique*.

non-functional testing: Testing the attributes of a component or system that do not relate to functionality, e.g. reliability, efficiency, usability, maintainability and portability.

O

off-the-shelf software: A software product that is developed for the general market, i.e. for a large number of customers, and that is delivered to many customers in identical format.

ATM **open source tool**: A software tool that is available to all potential users in source code form, usually via the internet; its users are permitted, usually under license, to study, change, improve and, at times, to distribute the software.

ATA **operability:** The capability of the software product to enable the user to operate and control it. [ISO 9126] See also *usability*.

ATT **operational acceptance testing:** Operational testing in the acceptance test phase, typically performed in a (simulated) operational environment by operations and/or systems administration staff focusing on operational aspects, e.g. recoverability, resource-behavior, installability and technical compliance. See also *operational testing.*

operational environment: Hardware and software products installed at users' or customers' sites where the component or system under test will be used. The software may include operating systems, database management systems, and other applications.

ETM ATT **operational profile**: The representation of a distinct set of tasks performed by the component or system, possibly based on user behavior when interacting with the component or system, and their probabilities of occurrence. A task is logical rather that physical and can be executed over several machines or be executed in non-contiguous time segments.

operational profile testing: Statistical testing using a model of system operations (short duration tasks) and their probability of typical use. [Musa]

ETM **operational profiling**: The process of developing and implementing an operational profile. See also *operational profile.*

operational testing: Testing conducted to evaluate a component or system in its operational environment. [IEEE 610]

oracle: See *test oracle*.

ATA **orthogonal array**: A 2-dimensional array constructed with special mathematical properties, such that choosing any two columns in the array provides every pair combination of each number in the array.

A-TA **orthogonal array testing:** A systematic way of testing all-pair combinations of variables using orthogonal arrays. It significantly reduces the number of all combinations of variables to test all pair combinations. See also *combinatorial testing, n-wise testing, pairwise testing.*

outcome: See *result*.

output: A variable (whether stored within a component or outside) that is written by a component.

output domain: The set from which valid output values can be selected. See also *domain*.

output value: An instance of an output. See also *output*.

outsourced testing: Testing performed by people who are not co-located with the project team and are not fellow employees.

P

pair programming: A software development approach whereby lines of code (production and/or test) of a component are written by two programmers sitting at a single computer. This implicitly means ongoing real-time code reviews are performed.

pair testing: Two persons, e.g. two testers, a developer and a tester, or an end-user and a tester, working together to find defects. Typically, they share one computer and trade control of it while testing.

ATT **pairwise integration testing:** A form of integration testing that targets pairs of components that work together, as shown in a call graph.

ATA **pairwise testing:** A black box test design technique in which test cases are designed to execute all possible discrete combinations of each pair of input parameters. See also *combinatorial testing, n-wise testing*, *orthogonal array testing.*

EITP **Pareto analysis:** A statistical technique in decision making that is used for selection of a limited number of factors that produce significant overall effect. In terms of quality improvement, a large majority of problems (80%) are produced by a few key causes (20%).

partition testing: See *equivalence partitioning*. [Beizer]

pass: A test is deemed to pass if its actual result matches its expected result.

pass/fail criteria: Decision rules used to determine whether a test item (function) or feature has passed or failed a test. [IEEE 829]

path: A sequence of events, e.g. executable statements, of a component or system from an entry point to an exit point.

path coverage: The percentage of paths that have been exercised by a test suite. 100% path coverage implies 100% LCSAJ coverage.

path sensitizing: Choosing a set of input values to force the execution of a given path.

ATT **path testing:** A white box test design technique in which test cases are designed to execute paths.

F **peer review:** A review of a software work product by colleagues of the producer of the product for the purpose of identifying defects and improvements. Examples are inspection, technical review and walkthrough.

performance: The degree to which a system or component accomplishes its designated functions within given constraints regarding processing time and throughput rate. [After IEEE 610] See also *efficiency*.

performance indicator: A high level metric of effectiveness and/or efficiency used to guide and control progressive development, e.g. lead-time slip for software development. [CMMI]

performance profiling: The task of analyzing, e.g., identifying performance bottlenecks based on generated metrics, and tuning the performance of a software component or system using tools.

F F-AT ATT **performance testing:** The process of testing to determine the performance of a software product. See also *efficiency testing*.

F ATT **performance testing tool:** A tool to support performance testing that usually has two main facilities: load generation and test transaction measurement. Load generation can simulate either multiple users or high volumes of input data. During execution, response time measurements are taken from selected transactions and these are logged. Performance testing tools normally provide reports based on test logs and graphs of load against response times.

ATA ATM **phase containment:** The percentage of defects that are removed in the same phase of the software lifecycle in which they were introduced.

phase test plan: A test plan that typically addresses one test phase. See also *test plan.*

ETM **planning poker**: A consensus-based estimation technique, mostly used to estimate effort or relative size of user stories in agile software development. It is a variation of the Wide Band Delphi method using a deck of cards with values representing the units in which the team estimates. See also *agile software development, Wide Band Delphi.*

pointer: A data item that specifies the location of another data item; for example, a data item that specifies the address of the next employee record to be processed. [IEEE 610]

portability: The ease with which the software product can be transferred from one hardware or software environment to another. [ISO 9126]

F ATT **portability testing:** The process of testing to determine the portability of a software product.

postcondition: Environmental and state conditions that must be fulfilled after the execution of a test or test procedure.

post-execution comparison: Comparison of actual and expected results, performed after the software has finished running.

post-project meeting: See *retrospective meeting.*

precondition: Environmental and state conditions that must be fulfilled before the component or system can be executed with a particular test or test procedure.

predicate: A statement that can evaluate to true or false and may be used to determine the control flow of subsequent decision logic. *See also* decision.

predicted outcome: See *expected result*.

pretest: See *intake test*.

ATM **priority:** The level of (business) importance assigned to an item, e.g. defect.

PRISMA (Product RISk MAnagement): A systematic approach to risk-based testing that employs product risk identification and analysis to create a product risk matrix based on likelihood and impact.

F **probe effect:** The effect on the component or system by the measurement instrument when the component or system is being measured, e.g. by a performance testing tool or monitor. For example performance may be slightly worse when performance testing tools are being used.

problem: See *defect*.

problem management: See *defect management*.

problem report: See *defect report*.

procedure testing: Testing aimed at ensuring that the component or system can operate in conjunction with new or existing users' business procedures or operational procedures.

process: A set of interrelated activities, which transform inputs into outputs. [ISO 12207]

EITP **process assessment:** A disciplined evaluation of an organization's software processes against a reference model. [after ISO 15504]

ETM **process-compliant testing**: Testing that follows a set of defined processes, e.g., defined by an external party such as a standards committee. See also *standard-compliant testing*.

process cycle test: A black box test design technique in which test cases are designed to execute business procedures and processes. [TMap] See also *procedure testing*.

ETAE **process-driven testing:** A scripting technique where scripts are structured into scenarios which represent use cases of the software under test. The scripts can be parameterized with test data.

process improvement: A program of activities designed to improve the performance and maturity of the organization's processes, and the result of such a program. [CMMI]

EITP **process model:** A framework wherein processes of the same nature are classified into a overall model, e.g. a test improvement model.

process reference model: A process model providing a generic body of best practices and how to improve a process in a prescribed step-by-step manner.

EITP **product-based quality:** A view of quality, wherein quality is based on a well-defined set of quality attributes. These attributes must be measured in an objective and quantitative way. Differences in the quality of products of the same type can be traced back to the way the specific quality attributes have been implemented. [After Garvin] See also *manufacturing-based quality, quality attribute, transcendent-based quality, user-based quality, value-based quality.*

F **product risk:** A risk directly related to the test object. See also *risk*.
F-AT
ATM
ATA
ATT

Product RISk MAnagement: See *PRISMA*.

production acceptance testing: See *operational acceptance testing.*

program instrumenter: See *instrumenter*.

program testing: See *component testing*.

project: A project is a unique set of coordinated and controlled activities with start and finish dates undertaken to achieve an objective conforming to specific requirements, including the constraints of time, cost and resources. [ISO 9000]

EITP **project retrospective:** A structured way to capture lessons learned and to create specific action plans for improving on the next project or next project phase.

F ATM **project risk:** A risk related to management and control of the (test) project, e.g. lack of staffing, strict deadlines, changing requirements, etc. See also *risk*.

project test plan: See *master test plan*.

pseudo-random: A series which appears to be random but is in fact generated according to some prearranged sequence.

Q

QFD: See *quality function deployment.*

qualification: The process of demonstrating the ability to fulfill specified requirements. Note the term 'qualified' is used to designate the corresponding status. [ISO 9000]

F **quality:** The degree to which a component, system or process meets specified requirements and/or user/customer needs and expectations. [After IEEE 610]

quality assurance: Part of quality management focused on providing confidence that quality requirements will be fulfilled. [ISO 9000]

quality attribute: A feature or characteristic that affects an item's quality. [IEEE 610]

quality characteristic: See *quality attribute*.

quality control: The operational techniques and activities, part of quality management, that are focused on fulfilling quality requirements. [after ISO 8402]

quality function deployment: A method to transform user demands into design quality, to deploy the functions forming quality, and to deploy methods for achieving the design quality into subsystems and component parts, and ultimately to specific elements of the manufacturing process. [Akao]

quality gate: A special milestone in a project. Quality gates are located between those phases of a project strongly depending on the outcome of a previous phase. A quality gate includes a formal check of the documents of the previous phase.

quality management: Coordinated activities to direct and control an organization with regard to quality. Direction and control with regard to quality generally includes the establishment of the quality policy and quality objectives, quality planning, quality control, quality assurance and quality improvement. [ISO 9000]

F-AT *ATM* **quality risk:** A product risk related to a quality attribute. See also *quality attribute, product risk*.

R

ETM **RACI matrix**: A matrix describing the participation by various roles in completing tasks or deliverables for a project or process. It is especially useful in clarifying roles and responsibilities. RACI is an acronym derived from the four key responsibilities most typically used: Responsible, Accountable, Consulted, and Informed.

random testing: A black box test design technique where test cases are selected, possibly using a pseudo-random generation algorithm, to match an operational profile. This technique can be used for testing non-functional attributes such as reliability and performance.

EITP **Rational Unified Process:** A proprietary adaptable iterative software development process framework consisting of four project lifecycle phases: inception, elaboration, construction and transition.

ETM **reactive testing:** Testing that dynamically responds to the actual system under test and test results being obtained. Typically reactive testing has a reduced planning cycle and the design and implementation test phases are not carried out until the test object is received.

recorder: See *scribe*.

ATT **record/playback tool:** See *capture/playback tool*.

recoverability: The capability of the software product to re-establish a specified level of performance and recover the data directly affected in case of failure. [ISO 9126] See also *reliability*.

ATT **recoverability testing:** The process of testing to determine the recoverability of a software product. See also *reliability testing*.

recovery testing: See *recoverability testing*.

ETM **regression-averse testing**: Testing using various techniques to manage the risk of regression, e.g., by designing re-usable testware and by extensive automation of testing at one or more test levels.

F F-AT ETAE **regression testing:** Testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a result of the changes made. It is performed when the software or its environment is changed.

regulation testing: See *compliance testing.*

release note: A document identifying test items, their configuration, current status and other delivery information delivered by development to testing, and possibly other stakeholders, at the start of a test execution phase. [After IEEE 829]

reliability: The ability of the software product to perform its required functions under stated conditions for a specified period of time, or for a specified number of operations. [ISO 9126]

ATT **reliability growth model:** A model that shows the growth in reliability over time during continuous testing of a component or system as a result of the removal of defects that result in reliability failures.

F ATT **reliability testing:** The process of testing to determine the reliability of a software product.

ATT **replaceability:** The capability of the software product to be used in place of another specified software product for the same purpose in the same environment. [ISO 9126] See also *portability*.

F **requirement:** A condition or capability needed by a user to solve a problem or achieve an objective that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document. [After IEEE 610]

ATA **requirements-based testing:** An approach to testing in which test cases are designed based on test objectives and test conditions derived from requirements, e.g. tests that exercise specific functions or probe non-functional attributes such as reliability or usability.

F **requirements management tool:** A tool that supports the recording of requirements, requirements attributes (e.g. priority, knowledge responsible) and annotation, and facilitates traceability through layers of requirements and requirements change management. Some requirements management tools also provide facilities for static analysis, such as consistency checking and violations to pre-defined requirements rules.

requirements phase: The period of time in the software lifecycle during which the requirements for a software product are defined and documented. [IEEE 610]

resource utilization: The capability of the software product to use appropriate amounts and types of resources, for example the amounts of main and secondary memory used by the program and the sizes of required temporary or overflow files, when the software performs its function under stated conditions. [After ISO 9126] See also *efficiency*.

ATT **resource utilization testing:** The process of testing to determine the resource-utilization of a software product. See also *efficiency testing*.

result: The consequence/outcome of the execution of a test. It includes outputs to screens, changes to data, reports, and communication messages sent out. See also *actual result, expected result.*

resumption criteria: The criteria used to restart all or a portion of the testing activities that were suspended previously.

resumption requirements: The defined set of testing activities that must be repeated when testing is re-started after a suspension. [After IEEE 829]

F ETAE **re-testing:** See *confirmation testing.*

EITP **retrospective meeting:** A meeting at the end of a project during which the project team members evaluate the project and learn lessons that can be applied to the next project.

F ATM **review:** An evaluation of a product or project status to ascertain discrepancies from planned results and to recommend improvements. Examples include management review, informal review, technical review, inspection, and walkthrough. [After IEEE 1028]

ATM **review plan**: A document describing the approach, resources and schedule of intended review activities. It identifies, amongst others: documents and code to be reviewed, review types to be used, participants, as well as entry and exit criteria to be applied in case of formal reviews, and the rationale for their choice. It is a record of the review planning process.

F **review tool:** A tool that provides support to the review process. Typical features include review planning and tracking support, communication support, collaborative reviews and a repository for collecting and reporting of metrics.

F ATM **reviewer:** The person involved in the review that identifies and describes anomalies in the product or project under review. Reviewers can be chosen to represent different viewpoints and roles in the review process.

F ATM ETAE **risk:** A factor that could result in future negative consequences; usually expressed as impact and likelihood.

ATM ATA ATT **risk analysis:** The process of assessing identified project or product risks to determine their level of risk, typically by estimating their impact and probability of occurrence (likelihood).

ATM ATT ETAE **risk assessment**: The process of identifying and subsequently analyzing the identified project or product risk to determine its level of risk, typically by assigning likelihood and impact ratings. *See also* product risk, project risk, risk, risk impact, risk level, risk likelihood.

F ATM ATA ATT **risk-based testing:** An approach to testing to reduce the level of product risks and inform stakeholders of their status, starting in the initial stages of a project. It involves the identification of product risks and the use of risk levels to guide the test process.

risk category: See *risk type*.

ATM ATA ATT **risk control:** The process through which decisions are reached and protective measures are implemented for reducing risks to, or maintaining risks within, specified levels.

ATM ATA **risk identification:** The process of identifying risks using techniques such as brainstorming, checklists and failure history. ATT

risk impact: The damage that will be caused if the risk become an actual outcome or event.

ATM ATA ATT **risk level:** The importance of a risk as defined by its characteristics impact and likelihood. The level of risk can be used to determine the intensity of testing to be performed. A risk level can be expressed either qualitatively (e.g. high, medium, low) or quantitatively.

risk likelihood: The estimated probability that a risk will become an actual outcome or event.

ATM ATA **risk management:** Systematic application of procedures and practices to the tasks of identifying, analyzing, prioritizing, and controlling risk.

ATM
ATA
ATT
ETAE

risk mitigation: See *risk control*.

risk type: A set of risks grouped by one or more common factors such as a quality attribute, cause, location, or potential effect of risk;. A specific set of product risk types is related to the type of testing that can mitigate (control) that risk type. For example the risk of user-interactions being misunderstood can be mitigated by usability testing.

ATT **robustness:** The degree to which a component or system can function correctly in the presence of invalid inputs or stressful environmental conditions. [IEEE 610] See also *error-tolerance, fault-tolerance.*

F **robustness testing:** Testing to determine the robustness of the software product.

ATM **root cause:** A source of a defect such that if it is removed, the occurrence of the defect type is decreased or removed. [CMMI]

ATA **root cause analysis:** An analysis technique aimed at identifying the root causes of defects. By directing corrective measures at root causes, it is hoped that the likelihood of defect recurrence will be minimized.

EITP **RUP:** See *Rational Unified Process*.

S

safety: The capability of the software product to achieve acceptable levels of risk of harm to people, business, software, property or the environment in a specified context of use. [ISO 9126]

safety critical system: A system whose failure or malfunction may result in death or serious injury to people, or loss or severe damage to equipment, or environmental harm.

safety testing: Testing to determine the safety of a software product.

sanity test: See *smoke test*.

scalability: The capability of the software product to be upgraded to accommodate increased loads. [After Gerrard]

scalability testing: Testing to determine the scalability of the software product.

scenario testing: See *use case testing*.

scorecard: A representation of summarized performance measurements representing progress towards the implementation of long-term goals. A scorecard provides static measurements of performance over or at the end of a defined interval. See also *balanced scorecard, dashboard.*

F **scribe:** The person who records each defect mentioned and any suggestions for process improvement during a review meeting, on a logging form. The scribe should ensure that the logging form is readable and understandable.

scripted testing: Test execution carried out by following a previously documented sequence of tests.

F **scripting language:** A programming language in which executable test scripts are written, used by a test execution tool (e.g. a capture/playback tool).

EITP **SCRUM:** An iterative incremental framework for managing projects commonly used with agile software development. See also *agile software development.*

security: Attributes of software products that bear on its ability to prevent unauthorized access, whether accidental or deliberate, to programs and data. [ISO 9126] See also *functionality.*

F
F-AT
ATT

security testing: Testing to determine the security of the software product. See also *functionality testing.*

security testing tool: A tool that provides support for testing security characteristics and vulnerabilities.

F **security tool:** A tool that supports operational security.

serviceability testing: See *maintainability testing*.

session-based test management: A method for measuring and managing session-based testing, e.g. exploratory testing.

session-based testing: An approach to testing in which test activities are planned as uninterrupted sessions of test design and execution, often used in conjunction with exploratory testing.

ATM **severity:** The degree of impact that a defect has on the development or operation of a component or system. [After IEEE 610]

ETM **Shewhart chart**: See *control chart.*

ATT **short-circuiting**: A programming language/interpreter technique for evaluating compound conditions in which a condition on one side of a logical operator may not be evaluated if the condition on the other side is sufficient to determine the final outcome.

simulation: The representation of selected behavioral characteristics of one physical or abstract system by another system. [ISO 2382/1]

simulator: A device, computer program or system used during testing, which behaves or operates like a given system when provided with a set of controlled inputs. [After IEEE 610, DO178b] See also *emulator*.

site acceptance testing: Acceptance testing by users/customers at their site, to determine whether or not a component or system satisfies the user/customer needs and fits within the business processes, normally including hardware as well as software.

ETM **S.M.A.R.T. goal methodology**: A methodology whereby objectives are defined very specifically rather than generically. SMART is an acronym derived from the attributes of the objective to be defined: Specific, Measurable, Attainable, Relevant and Timely.

smoke test: A subset of all defined/planned test cases that cover the main functionality of a component or system, to ascertaining that the most crucial functions of a program work, but not bothering with finer details. See also *build, verification test, intake test*.

software: Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system. [IEEE 610]

software attack: *See attack.*

Software Failure Mode and Effect Analysis (SFMEA): See *Failure Mode and Effect Analysis (FMEA).*

Software Failure Mode, Effects, and Criticality Analysis (SFMECA): See *Failure Mode, Effects, and Criticality Analysis (FMECA).*

Software Fault Tree Analysis (SFTA): See *Fault Tree Analysis (FTA).*

software feature: See *feature*.

software integrity level: The degree to which software complies or must comply with a set of stakeholder-selected software and/or software-based system characteristics (e.g., software complexity, risk assessment, safety level, security level, desired performance, reliability, or cost) which are defined to reflect the importance of the software to its stakeholders.

F-AT EITP **software lifecycle:** The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software lifecycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and sometimes, retirement phase. Note these phases may overlap or be performed iteratively.

EITP **Software Process Improvement:** A program of activities designed to improve the performance and maturity of the organization's software processes and the results of such a program. [After CMMI]

software product characteristic: See *quality attribute.*

software quality: The totality of functionality and features of a software product that bear on its ability to satisfy stated or implied needs. [After ISO 9126] See also *quality*.

software quality characteristic: See *quality attribute*.

software test incident: See *incident*.

software test incident report: See *incident report*.

ATA **Software Usability Measurement Inventory (SUMI):** A questionnaire-based usability test technique for measuring software quality from the end user's point of view. [Veenendaal04]

source statement: See *statement*.

specification: A document that specifies, ideally in a complete, precise and verifiable manner, the requirements, design, behavior, or other characteristics of a component or system, and, often, the procedures for determining whether these provisions have been satisfied. [After IEEE 610]

ATA **specification-based techniqu**e: See *black box test design technique.*

specification-based testing: See *black box testing*.

specification-based test design technique: See *black box test design technique*.

specified input: An input for which the specification predicts a result.

EITP **SPI:** See *Software Process Improvement.*

ATT **stability:** The capability of the software product to avoid unexpected effects from modifications in the software. [ISO 9126] See also *maintainability*.

EITP **staged representation:** A model structure wherein attaining the goals of a set of process areas establishes a maturity level; each level builds a foundation for subsequent levels. [CMMI]

EITP **standard:** Formal, possibly mandatory, set of requirements developed and used to prescribe consistent approaches to the way of working or to provide guidelines (e.g., ISO/IEC standards, IEEE standards, and organizational standards). [After CMMI]

ETM **standard-compliant testing**: Testing that complies to a set of requirements defined by a standard, e.g., an industry testing standard or a standard for testing safety-critical systems. See also *process-compliant testing*.

standard software: See *off-the-shelf software*.

standards testing: See *compliance testing*.

state diagram: A diagram that depicts the states that a component or system can assume, and shows the events or circumstances that cause and/or result from a change from one state to another. [IEEE 610]

state table: A grid showing the resulting transitions for each state combined with each possible event, showing both valid and invalid transitions.

state transition: A transition between two states of a component or system.

F ATA **state transition testing:** A black box test design technique in which test cases are designed to execute valid and invalid state transitions. See also *N-switch testing*.

statement: An entity in a programming language, which is typically the smallest indivisible unit of execution.

F **statement coverage:** The percentage of executable statements that have been exercised by a test suite.

ATT **statement testing:** A white box test design technique in which test cases are designed to execute statements.

F ATT **static analysis:** Analysis of software development artifacts, e.g. requirements or code, carried out without execution of these software development artifacts. Static analysis is usually carried out by means of a supporting tool.

F **static analysis tool:** See *static analyzer*.

ATT **static analyzer:** A tool that carries out static analysis.

static code analysis: Analysis of source code carried out without execution of that software.

static code analyzer: A tool that carries out static code analysis. The tool checks source code, for certain properties such as conformance to coding standards, quality metrics or data flow anomalies.

F **static testing:** Testing of a software development artifact, e.g., requirements, design or code, without execution of these artifacts, e.g., reviews or static analysis.

statistical testing: A test design technique in which a model of the statistical distribution of the input is used to construct representative test cases. See also *operational profile testing*.

status accounting: An element of configuration management, consisting of the recording and reporting of information needed to manage a configuration effectively. This information includes a listing of the approved configuration identification, the status of proposed changes to the configuration, and the implementation status of the approved changes. [IEEE 610]

ATM EITP **STEP:** See *Systematic Test and Evaluation Process.*

storage: See *resource utilization.*

storage testing: See *resource utilization testing*.

F **stress testing:** A type of performance testing conducted to evaluate a system or component at or beyond the limits of its anticipated or specified workloads, or with reduced availability of resources such as access to memory or servers. [After IEEE 610] See also *performance testing, load testing*.

F **stress testing tool:** A tool that supports stress testing.

structural coverage: Coverage measures based on the internal structure of a component or system.

structural test design technique: See *white-box test design technique*.

F **structural testing:** See *white-box testing*.

structure-based test design technique: See *white-box test design technique*.

ATT **structure-based technique:** See *white-box test design technique*.

F **structure-based testing:** See *white-box testing*.

ETAE **structured scripting:** A scripting technique that builds and utilizes a library of reusable (parts of) scripts.

structured walkthrough: See *walkthrough*.

F ETAE **stub:** A skeletal or special-purpose implementation of a software component, used to develop or test a component that calls or is otherwise dependent on it. It replaces a called component. [After IEEE 610]

subpath: A sequence of executable statements within a component.

suitability: The capability of the software product to provide an appropriate set of functions for specified tasks and user objectives. [ISO 9126] See also *functionality*.

ATA **suitability testing:** The process of testing to determine the suitability of a software product

ATA **SUMI:** See *Software Usability Measurement Inventory*.

suspension criteria: The criteria used to (temporarily) stop all or a portion of the testing activities on the test items. [After IEEE 829]

ETAE **SUT:** Acronym for system under test.

syntax testing: A black box test design technique in which test cases are designed based upon the definition of the input domain and/or output domain.

system: A collection of components organized to accomplish a specific function or set of functions. [IEEE 610]

system integration testing: Testing the integration of systems and packages; testing interfaces to external organizations (e.g. Electronic Data Interchange, Internet).

system of systems: Multiple heterogeneous, distributed systems that are embedded in networks at multiple levels and in multiple interconnected domains, addressing large-scale inter-disciplinary common problems and purposes, usually without a common management structure.

system under test: See *test object*.

F **system testing:** The process of testing an integrated system to verify that it meets specified requirements. [Hetzel]

ATM EITP **Systematic Test and Evaluation Process:** A structured testing methodology, also used as a content-based model for improving the testing process. Systematic Test and Evaluation Process (STEP) does not require that improvements occur in a specific order. See also *content-based model.*

T

TDD: *See* test-driven development.

F ATM **technical review:** A peer group discussion activity that focuses on achieving consensus on the technical approach to be taken. [Gilb and Graham], [IEEE 1028] See also *peer review*.

test: A set of one or more test cases. [IEEE 829]

ETAE **test adaption layer:** The layer in a generic test automation architecture which provides the necessary code to adapt the automated tests for the various components, configuration or interfaces of the SUT.

test analysis: The process of analyzing the test basis and defining test objectives.

F F-AT ATM **test approach:** The implementation of the test strategy for a specific project. It typically includes the decisions made that follow based on the (test) project's goal and the risk assessment carried out, starting points regarding the test process, the test design techniques to be applied, exit criteria and test types to be performed

ETM **test architect :** (1) A person who provides guidance and strategic direction for a test organization and for its relationship with other disciplines.
(2) A person who defines the way testing is structured for a given system, including topics such as test tools and test data management.

F-AT **test automation:** The use of software to perform or support test activities, e.g. test management, test design, test execution and results checking.

ETAE **test automation architecture:** An instantiation of the generic test automation architecture to define the architecture of a test automation solution, i.e., its layers, components, services and interfaces.

ETAE **test automation engineer**: A person who is responsible for the design, implementation and maintenance of a test automation architecture as well as the technical evolution of the resulting test automation solution.

ETAE **test automation framework**: A tool that provides an environment for test automation. It usually includes a test harness and test libraries.

ETAE **test automation manager**: A person who is responsible for the planning and supervision of the development and evolution of a test automation solution.

ETAE **test automation solution**: A realization/implementation of a test automation architecture, i.e., a combination of components implementing a specific test automation assignment. The components may include off-the-shelf test tools, test automation frameworks, as well as test hardware.

ETAE **test automation strategy**: A high-level plan to achieve long-term objectives of test automation under given boundary conditions.

F F-AT **test basis:** All documents from which the requirements of a component or system can be inferred. The documentation on which the test cases are based. If a document can be amended only by way of formal amendment procedure, then the test basis is called a frozen test basis. [After TMap]

test bed: See *test environment*.

F ATM **test case:** A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement. [After IEEE 610]

test case design technique: See *test design technique*.

ETAE **test case result:** The final verdict on the execution of a test and its outcomes, like pass, fail, or error. The result of error is used for situations where it is not clear whether the problem is in the test object.

F **test case specification:** A document specifying a set of test cases (objective, inputs, test actions, expected results, and execution preconditions) for a test item. [After IEEE 829] See also *test specification.*

test case suite: See *test suite*.

F-AT ATA **test charter:** A statement of test objectives, and possibly test ideas about how to test. Test charters are used in exploratory testing. See also *exploratory testing*.

ATM **test closure:** During the test closure phase of a test process data is collected from completed activities to consolidate experience, testware, facts and numbers. The test closure phase consists of finalizing and archiving the testware and evaluating the test process, including preparation of a test evaluation report. See also *test process.*

F **test comparator:** A test tool to perform automated test comparison of actual results with expected results.

test comparison: The process of identifying differences between the actual results produced by the component or system under test and the expected results for a test. Test comparison can be performed during test execution (dynamic comparison) or after test execution.

test completion criteria: See *exit criteria*.

F ATM **test condition:** An item or event of a component or system that could be verified by one or more test cases, e.g. a function, transaction, feature, quality attribute, or structural element.

F ATM ATA **test control:** A test management task that deals with developing and applying a set of corrective actions to get a test project on track when monitoring shows a deviation from what was planned. See also *test management*.

F **test coverage:** See *coverage*.

test cycle: Execution of the test process against a single identifiable release of the test object.

F **test data:** Data that exists (for example, in a database) before a test is executed, and that affects or is affected by the component or system under test.

test data management: The process of analyzing test data requirements, designing test data structures, creating and maintaining test data.

F ATA **test data preparation tool:** A type of test tool that enables data to be selected from existing databases or created, generated, manipulated and edited for use in testing.

ETAE **test definition layer:** The layer in a generic test automation architecture which supports test implementation by supporting the definition of test suites and/or test cases, e.g., by offering templates or guidelines.

test deliverable: Any test (work) product that must be delivered to someone other than the test (work) product's author. See also *deliverable.*

F ATM ATA **test design:** (1) See *test design specification*.
(2) The process of transforming general test objectives into tangible test conditions and test cases.

test design specification: A document specifying the test conditions (coverage items) for a test item, the detailed test approach and identifying the associated high level test cases. [After IEEE 829] See also *test specification.*

F **test design technique:** Procedure used to derive and/or select test cases.

F ATA **test design tool:** A tool that supports the test design activity by generating test inputs from a specification that may be held in a CASE tool repository, e.g. requirements management tool, from specified test conditions held in the tool itself, or from code.

ATM **test director:** A senior manager who manages test managers. *See also* test manager.

F F-AT ETM **test-driven development:** A way of developing software where the test cases are developed, and often automated, before the software is developed to run those test cases.

test driver: See *driver*.

F **test environment:** An environment containing hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test. [After IEEE 610]

F-AT ATM **test estimation:** The calculated approximation of a result related to various aspects of testing (e.g. effort spent, completion date, costs involved, number of test cases, etc.) which is usable even if input data may be incomplete, uncertain, or noisy.

test evaluation report: A document produced at the end of the test process summarizing all testing activities and results. It also contains an evaluation of the test process and lessons learned.

F ATM ATA **test execution:** The process of running a test on the component or system under test, producing actual result(s).

F-AT ETAE **test execution automation:** The use of software, e.g. capture/playback tools, to control the execution of tests, the comparison of actual results to expected results, the setting up of test preconditions, and other test control and reporting functions.

ETAE **test execution layer:** The layer in a generic test automation architecture which supports the execution of test suites and/or test cases.

test execution phase: The period of time in a software development lifecycle during which the components of a software product are executed, and the software product is evaluated to determine whether or not requirements have been satisfied. [IEEE 610]

F **test execution schedule:** A scheme for the execution of test procedures. Note: The test procedures are included in the test execution schedule in their context and in the order in which they are to be executed.

test execution technique: The method used to perform the actual test execution, either manual or automated.

F ATA ATT ETAE **test execution tool:** A type of test tool that is able to execute other software using an automated test script, e.g. capture/playback. [Fewster and Graham]

test fail: See *fail*.

ETAE **test generation layer:** The layer in a generic test automation architecture which supports manual or automated design of test suites and/or test cases.

test generator: See *test data preparation tool*.

F **test harness:** A test environment comprised of stubs and drivers needed to execute a test.

ETAE **test hook:** A customized software interface that enables automated testing of a test object.

ATM ATA **test implementation:** The process of developing and prioritizing test procedures, creating test data and, optionally, preparing test harnesses and writing automated test scripts.

EITP **test improvement plan:** A plan for achieving organizational test process improvement objectives based on a thorough understanding of the current strengths and weaknesses of the organization's test processes and test process assets. [After CMMI]

test incident: See *incident*.

test incident report: See *incident report*.

test infrastructure: The organizational artifacts needed to perform testing, consisting of test environments, test tools, office environment and procedures.

test input: The data received from an external source by the test object during test execution. The external source can be hardware, software or human.

test item: The individual element to be tested. There usually is one test object and many test items. See also *test object*.

test item transmittal report: See *release note*.

F **test leader:** See *test manager.*

F ATM **test level:** A group of test activities that are organized and managed together. A test level is linked to the responsibilities in a project. Examples of test levels are component test, integration test, system test and acceptance test. [After TMap]

F ATM **test log:** A chronological record of relevant details about the execution of tests. [IEEE 829]

ETAE **test logging:** The process of recording information about tests executed into a test log.

ATM **test management:** The planning, estimating, monitoring and control of test activities, typically carried out by a test manager.

F ATT **test management tool:** A tool that provides support to the test management and control part of a test process. It often has several capabilities, such as testware management, scheduling of tests, the logging of results, progress tracking, incident management and test reporting.

F **test manager:** The person responsible for project management of testing activities and resources, and evaluation of a test object. The individual who directs, controls, administers, plans and regulates the evaluation of a test object.

ATM EITP **Test Maturity Model integration:** A five level staged framework for test process improvement, related to the Capability Maturity Model Integration (CMMI), that describes the key elements of an effective test process.

ETM **test mission**: The purpose of testing for an organization, often documented as part of the test policy. *See also* test policy.

F ATM **test monitoring:** A test management task that deals with the activities related to periodically checking the status of a test project. Reports are prepared that compare the actuals to that

ATA which was planned. See also *test management*.

test object: The component or system to be tested. See also *test item*.

F **test objective:** A reason or purpose for designing and executing a test.

F-AT **test oracle:** A source to determine expected results to compare with the actual result of the software under test. An oracle may be the existing system (for a benchmark), other software, a user manual, or an individual's specialized knowledge, but should not be the code. [After Adrion]

test outcome: See *result*.

test pass: See *pass*.

test performance indicator: A high level metric of effectiveness and/or efficiency used to guide and control progressive test development, e.g. Defect Detection Percentage (DDP).

test phase: A distinct set of test activities collected into a manageable phase of a project, e.g. the execution activities of a test level. [After Gerrard]

F ATM **test plan:** A document describing the scope, approach, resources and schedule of intended test activities. It identifies amongst others test items, the features to be tested, the testing tasks, who will do each task, degree of tester independence, the test environment, the test design techniques and entry and exit criteria to be used, and the rationale for their choice, and any risks requiring contingency planning. It is a record of the test planning process. [After IEEE 829]

ATM ATA **test planning:** The activity of establishing or updating a test plan.

Test Point Analysis (TPA): A formula based test estimation method based on function point analysis. [TMap]

F ATM EITP ETM **test policy:** A high level document describing the principles, approach and major objectives of the organization regarding testing.

F ATM **test procedure:** See *test procedure specification*.

F ATM **test procedure specification:** A document specifying a sequence of actions for the execution of a test. Also known as test script or manual test script. [After IEEE 829] See also *test specification.*

test process: The fundamental test process comprises test planning and control, test analysis and design, test implementation and execution, evaluating exit criteria and reporting, and test closure activities.

EITP **Test Process Group:** A collection of (test) specialists who facilitate the definition, maintenance, and improvement of the test processes used by an organization. [After CMMI]

ETAE **test process improvement**: A program of activities designed to improve the performance and maturity of the organization's test processes and the results of such a program. [After CMMI]

EITP **test process improvement manifesto:** A statement that echoes the agile manifesto, and defines values for improving the testing process. The values are:
- flexibility over detailed processes
- best practices over templates
- deployment orientation over process orientation
- peer reviews over quality assurance (departments)
- business driven over model driven. [Veenendaal08]

EITP **test process improver:** A person implementing improvements in the test process based on a test improvement plan.

test progress report: A document summarizing testing activities and results, produced at regular intervals, to report progress of testing activities against a baseline (such as the original test plan) and to communicate risks and alternatives requiring a decision to management.

test record: See *test log*.

test recording: See *test logging*.

test report: See *test summary report* and *test progress report*.

ETAE **test reporting:** Collecting and analyzing data from testing activities and subsequently consolidating the data in a report to inform stakeholders. See also test process.

test reproducibility: An attribute of a test indicating whether the same results are produced each time the test is executed.

test requirement: See *test condition*.

test result: See *result*.

test rig: See *test environment.*

test run: Execution of a test on a specific version of the test object.

test run log: See *test log*.

test scenario: See *test procedure specification.*

test schedule: A list of activities, tasks or events of the test process, identifying their intended start and finish dates and/or times, and interdependencies.

F ATM ETAE **test script:** Commonly used to refer to a test procedure specification, especially an automated one.

test session: An uninterrupted period of time spent in executing tests. In exploratory testing, each test session is focused on a charter, but testers can also explore new opportunities or issues during a session. The tester creates and executes on the fly and records their progress. See also *exploratory testing*.

test set: See *test suite*.

test situation: See *test condition.*

test specification: A document that consists of a test design specification, test case specification and/or test procedure specification.

test specification technique: See *test design technique.*

test stage: See *test level*.

F F-AT ATM ATA ETM **test strategy:** A high-level description of the test levels to be performed and the testing within those levels for an organization or programme (one or more projects).

F **test suite:** A set of several test cases for a component or system under test, where the post condition of one test is often used as the precondition for the next one.

F ATM **test summary report:** A document summarizing testing activities and results. It also contains an evaluation of the corresponding test items against exit criteria. [After IEEE 829]

test target: A set of exit criteria.

test technique: See *test design technique.*

EITP **test tool:** A software product that supports one or more test activities, such as planning and control, specification, building initial files and data, test execution and test analysis. [TMap] See also *CAST*.

test type: A group of test activities aimed at testing a component or system focused on a specific test objective, i.e. functional test, usability test, regression test etc. A test type may take place on one or more test levels or test phases. [After TMap]

ATT ETAE **testability:** The capability of the software product to enable modified software to be tested. [ISO 9126] See also *maintainability*.

testability review: A detailed check of the test basis to determine whether the test basis is at an adequate quality level to act as an input document for the test process. [After TMap]

testable requirement: A requirements that is stated in terms that permit establishment of test designs (and subsequently test cases) and execution of tests to determine whether the requirement has been met. [After IEEE 610]

F **tester:** A skilled professional who is involved in the testing of a component or system.

F **testing:** The process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.

F ETAE **testware:** Artifacts produced during the test process required to plan, design, and execute tests, such as documentation, scripts, inputs, expected results, set-up and clear-up procedures, files, databases, environment, and any additional software or utilities used in testing. [After Fewster and Graham]

thread testing: An approach to component integration testing where the progressive integration of components follows the implementation of subsets of the requirements, as opposed to the integration of components by levels of a hierarchy.

three point estimation: A test estimation method using estimated values for the “best case”, “worst case”, and “most likely case” of the matter being estimated, to define the degree of certainty associated with the resultant estimate.

time behavior: See *performance*.

ATM EITP **TMMi:** See *Test Maturity Model integration.*

top-down testing: An incremental approach to integration testing where the component at the top of the component hierarchy is tested first, with lower level components being simulated by stubs. Tested components are then used to test lower level components. The process is repeated until the lowest level components have been tested. See also *integration testing.*

EITP **Total Quality Management:** An organization-wide management approach centered on quality, based on the participation of all members of the organization and aiming at long-term success through customer satisfaction, and benefits to all members of the organization and to society. Total Quality Management consists of planning, organizing, directing, control, and assurance. [After ISO 8402]

EITP **TPG:** See *Test Process Group.*

ATM EITP **TPI Next:** A continuous business-driven framework for test process improvement that describes the key elements of an effective and efficient test process.

EITP **TQM:** See *Total Quality Management.*

F **traceability:** The ability to identify related items in documentation and software, such as requirements with associated tests. See also *horizontal traceability*, *vertical traceability*.

traceability matrix: A two-dimensional table, which correlates two entities (e.g., requirements and test cases). The table allows tracing back and forth the links of one entity to the other, thus enabling the determination of coverage achieved and the assessment of impact of proposed changes.

EITP **transactional analysis:** The analysis of transactions between people and within people’s minds; a transaction is defined as a stimulus plus a response. Transactions take place between people and between the ego states (personality segments) within one person’s mind.

EITP **transcendent-based quality:** A view of quality, wherein quality cannot be precisely defined, but we know it when we see it, or are aware of its absence when it is missing. Quality depends on the perception and affective feelings of an individual or group of individuals towards a product. [After Garvin] See also *manufacturing-based quality, product-based quality, user-based quality, value-based quality.*

U

ATA **understandability:** The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use. [ISO 9126] See also *usability*.

unit: See *component*.

F F-AT **unit test framework:** A tool that provides an environment for unit or component testing in which a component can be tested in isolation or with suitable stubs and drivers. It also provides other support for the developer, such as debugging capabilities. [Graham]

unit testing: See *component testing*.

unreachable code: Code that cannot be reached and therefore is impossible to execute.

usability: The capability of the software to be understood, learned, used and attractive to the user when used under specified conditions. [ISO 9126]

F F-AT ATA **usability testing:** Testing to determine the extent to which the software product is understood, easy to learn, easy to operate and attractive to the users under specified conditions. [After ISO 9126]

use case: A sequence of transactions in a dialogue between an actor and a component or system with a tangible result, where an actor can be a user or anything that can exchange information with the system.

F ATA **use case testing:** A black box test design technique in which test cases are designed to execute scenarios of use cases.

F **user acceptance testing:** See *acceptance testing*.

EITP **user-based quality:** A view of quality, wherein quality is the capacity to satisfy needs, wants and desires of the user(s). A product or service that does not fulfill user needs is unlikely to find any users. This is a context dependent, contingent approach to quality since different business characteristics require different qualities of a product. [after Garvin] See also *manufacturing-based quality, product-based quality, transcendent-based quality, value-based quality.*

user scenario testing: See *use case testing*

F-AT **user story:** A high-level user or business requirement commonly used in agile software development, typically consisting of one or more sentences in the everyday or business language capturing what functionality a user needs, any non-functional criteria, and also includes acceptance criteria. See also *agile software development, requirement*.

ATA **user story testing:** A black box test design technique in which test cases are designed based on user stories to verify their correct implementation. See also *user story*.

user test: A test whereby real-life users are involved to evaluate the usability of a component or system.

V

F **V-model:** A framework to describe the software development lifecycle activities from requirements specification to maintenance. The V-model illustrates how testing activities can be integrated into each phase of the software development lifecycle.

F **validation:** Confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled. [ISO 9000]

EITP **value-based quality:** A view of quality, wherein quality is defined by price. A quality product or service is one that provides desired performance at an acceptable cost. Quality is determined by means of a decision process with stakeholders on trade-offs between time, effort and cost aspects. [After Garvin] See also *manufacturing-based quality, product-based quality, transcendent-based quality, user-based quality*.

variable: An element of storage in a computer that is accessible by a software program by referring to it by a name.

F ETAE **verification:** Confirmation by examination and through provision of objective evidence that specified requirements have been fulfilled. [ISO 9000]

F **version control:** See *configuration control.*

vertical traceability: The tracing of requirements through the layers of development documentation to components.

volume testing: Testing where the system is subjected to large volumes of data. See also *resource-utilization testing*.

W

F ATM **walkthrough:** A step-by-step presentation by the author of a document in order to gather information and to establish a common understanding of its content. [Freedman and Weinberg, IEEE 1028] See also *peer review*.

ATA **WAMMI**: See *Website Analysis and MeasureMent Inventory.*

WBS: See *Work Breakdown Structure.*

ATA **Website Analysis and MeasureMent Inventory (WAMMI):** A questionnaire-based usability test technique for measuring web site software quality from the end user's point of view.

white-box technique: See *white-box test design technique.*

F ATT **white-box test design technique:** Procedure to derive and/or select test cases based on an analysis of the internal structure of a component or system.

F **white-box testing:** Testing based on an analysis of the internal structure of the component or system.

ATM **Wide Band Delphi:** An expert based test estimation technique that aims at making an accurate estimation using the collective wisdom of the team members.

ATT **wild pointer:** A pointer that references a location that is out of scope for that pointer or that does not exist. See also *pointer.*

Work Breakdown Structure: An arrangement of work elements and their relationship to each other and to the end product. [CMMI]

Annex A: References

Standards

[DO-178b] DO-178B:1992. Software Considerations in Airborne Systems and Equipment Certification, Requirements and Technical Concepts for Aviation (RTCA SC167)

[IEEE 610] IEEE 610.12:1990. Standard Glossary of Software Engineering Terminology.

[IEEE 829] IEEE 829:1998. Standard for Software Test Documentation

[IEEE 1008] IEEE 1008:1993. Standard for Software Unit Testing

[IEEE 1028] IEEE 1028:1997. Standard for Software Reviews and Audits

[IEEE 1044] IEEE 1044:1993. Standard Classification for Software Anomalies

[IEEE 1219] IEEE 1219:1998. Software Maintenance

[ISO 2382/1] ISO/IEC 2382-1:1993. Data processing - Vocabulary - Part 1: Fundamental terms

[ISO 8402] ISO 8402: 1994. Quality Management and Quality Assurance Vocabulary

[ISO 9000] ISO 9000:2005. Quality Management Systems – Fundamentals and Vocabulary

[ISO 9126] ISO/IEC 9126-1:2001. Software Engineering – Software Product Quality – Part 1: Quality characteristics and sub-characteristics

[ISO 12207] ISO/IEC 12207:1995. Information Technology – Software Lifecycle Processes

[ISO 14598] ISO/IEC 14598-1:1999. Information Technology – Software Product Evaluation - Part 1: General Overview

[ISO 15504] ISO/IEC 15504-9: 1998. Information Technology – Software Process Assessment – Part 9: Vocabulary

Books and Papers

[Abbott] J. Abbot (1986), *Software Testing Techniques*, NCC Publications

[Adrion] W. Adrion, M. Branstad and J. Cherniabsky (1982), Validation, Verification and Testing of Computer Software, in: *Computing Surveys*, Vol. 14, No 2, June 1982

[Akao] Akao, Yoji (1994), *Development History of Quality Function Deployment - The Customer Driven Approach to Quality Planning and Deployment*, Minato, Tokyo 107 Japan: Asian Productivity Organization, pp. 339, ISBN 92-833-1121-3

[Bach] J. Bach (2004), Exploratory Testing, in: E. van Veenendaal, *The Testing Practitioner – 2nd edition*, UTN Publishing, ISBN 90-72194-65-9

[Beizer] B. Beizer (1990), *Software Testing Techniques*, van Nostrand Reinhold, ISBN 0-442-20672-0

[Chow] T. Chow (1978), Testing Software Design Modelled by Finite-Sate Machines, in: *IEEE Transactions on Software Engineering*, Vol. 4, No 3, May 1978

[CMM] M. Paulk, C. Weber, B. Curtis and M.B. Chrissis (1995), *The Capability Maturity Model, Guidelines for Improving the Software Process*, Addison-Wesley, ISBN 0-201-54664-7

[CMMI] M.B. Chrissis, M. Konrad and S. Shrum (2004), *CMMI, Guidelines for Process Integration and Product Improvement*, Addison Wesley, ISBN 0-321-15496-7

[Deming] D. W. Edwards (1986), *Out of the Crisis*, MIT Center for Advanced Engineering Study, ISBN 0-911379-01-0

[Fenton] N. Fenton (1991), *Software Metrics: a Rigorous Approach*, Chapman & Hall, ISBN 0-53249-425-1

[Fewster and Graham] M. Fewster and D. Graham (1999), *Software Test Automation, Effective use of test execution tools*, Addison-Wesley, ISBN 0-201-33140-3

[Freedman and Weinberg] D. Freedman and G. Weinberg (1990), *Walkthroughs, Inspections, and Technical Reviews,* Dorset House Publishing, ISBN 0-932633-19-6

[Garvin] D.A. Garvin (1984), What does product quality really mean?, in: *Sloan Management Review*, Vol. 26, nr. 1 1984

[Gerrard] P. Gerrard and N. Thompson (2002), *Risk-Based E-Business Testing,* Artech House Publishers, ISBN 1-58053-314-0

[Gilb and Graham] T. Gilb and D. Graham (1993), *Software Inspection*, Addison-Wesley, ISBN 0-201-63181-4

[Graham] D. Graham, E. van Veenendaal, I. Evans and R. Black (2007), *Foundations of Software Testing, Thomson Learning*, ISBN 978-1-84480-355-2

[Grochtmann] M. Grochtmann (1994), Test Case Design Using Classification Trees, in: *Conference Proceedings STAR* 1994

[Hetzel] W. Hetzel (1988), *The complete guide to software testing – 2nd edition*, QED Information Sciences, ISBN 0-89435-242-3

[Juran] J.M. Juran (1979), *Quality Control Handbook,* McGraw-Hill

[McCabe] T. McCabe (1976), A complexity measure, in: *IEEE Transactions on Software Engineering,* Vol. 2, pp. 308-320

[Musa] J. Musa (1998), *Software Reliability Engineering Testing*, McGraw-Hill Education, ISBN 0-07913-271-5

[Myers] G. Myers (1979), *The Art of Software Testing*, Wiley, ISBN 0-471-04328-1

[TMap] M. Pol, R. Teunissen, E. van Veenendaal (2002), *Software Testing, A guide to the TMap Approach,* Addison Wesley, ISBN 0-201-745712

[TMMi] E. van Veenendaal and J. Cannegieter (2011), *The Little TMMi*, UTN Publishing, ISBN 97-89490986-03-2

[Veenendaal04] E. van Veenendaal (2004), *The Testing Practitioner – 2nd edition,* UTN Publishing, ISBN 90-72194-65-9

[Veenendaal08] E. van Veenendaal (2008), Test Improvement Manifesto, in: *Testing Experience*, Issue 04/08, December 2008

ISTQB Foundation Extension Agile Tester

Syllabus

Certified Tester

Foundation Level Extension Syllabus
Agile Tester

Version 2014

International Software Testing Qualifications Board

Foundation Level Extension Agile Tester Working Group: Rex Black (Chair), Bertrand Cornanguer (Vice Chair), Gerry Coleman (Learning Objectives Lead), Debra Friedenberg (Exam Lead), Alon Linetzki (Business Outcomes and Marketing Lead), Tauhida Parveen (Editor), and Leo van der Aalst (Development Lead).

Authors: Rex Black, Anders Claesson, Gerry Coleman, Bertrand Cornanguer, Istvan Forgacs, Alon Linetzki, Tilo Linz, Leo van der Aalst, Marie Walsh, and Stephan Weber.

Internal Reviewers: Mette Bruhn-Pedersen, Christopher Clements, Alessandro Collino, Debra Friedenberg, Kari Kakkonen, Beata Karpinska, Sammy Kolluru, Jennifer Leger, Thomas Mueller, Tuula Pääkkönen, Meile Posthuma, Gabor Puhalla, Lloyd Roden, Marko Rytkönen, Monika Stoecklein-Olsen, Robert Treffny, Chris Van Bael, and Erik van Veenendaal; 2013-2014.

Revision History

Version	Date	Remarks
Syllabus v0.1	26JUL2013	Standalone sections
Syllabus v0.2	16SEP2013	WG review comments on v01 incorporated
Syllabus v0.3	20OCT2013	WG review comments on v02 incorporated
Syllabus v0.7	16DEC2013	Alpha review comments on v03 incorporated
Syllabus v0.71	20DEC2013	Working group updates on v07
Syllabus v0.9	30JAN2014	Beta version
Syllabus 2014	31MAY2014	GA version

Table of Contents

Acknowledgements

This document was produced by a team from the International Software Testing Qualifications Board Foundation Level Working Group.

The Agile Extension team thanks the review team and the National Boards for their suggestions and input.

At the time the Foundation Level Agile Extension Syllabus was completed, the Agile Extension Working Group had the following membership: Rex Black (Chair), Bertrand Cornanguer (Vice Chair), Gerry Coleman (Learning Objectives Lead), Debra Friedenberg (Exam Lead), Alon Linetzki (Business Outcomes and Marketing Lead), Tauhida Parveen (Editor), and Leo van der Aalst (Development Lead).

Authors: Rex Black, Anders Claesson, Gerry Coleman, Bertrand Cornanguer, Istvan Forgacs, Alon Linetzki, Tilo Linz, Leo van der Aalst, Marie Walsh, and Stephan Weber.

Internal Reviewers: Mette Bruhn-Pedersen, Christopher Clements, Alessandro Collino, Debra Friedenberg, Kari Kakkonen, Beata Karpinska, Sammy Kolluru, Jennifer Leger, Thomas Mueller, Tuula Pääkkönen, Meile Posthuma, Gabor Puhalla, Lloyd Roden, Marko Rytkönen, Monika Stoecklein-Olsen, Robert Treffny, Chris Van Bael, and Erik van Veenendaal.

The team thanks also the following persons, from the National Boards and the Agile expert community, who participated in reviewing, commenting, and balloting of the Foundation Agile Extension Syllabus: Dani Almog, Richard Berns, Stephen Bird, Monika Bögge, Afeng Chai, Josephine Crawford, Tibor Csöndes, Huba Demeter, Arnaud Foucal, Cyril Fumery, Kobi Halperin, Inga Hansen, Hanne Hinz, Jidong Hu, Phill Isles, Shirley Itah, Martin Klonk, Kjell Lauren, Igal Levi, Rik Marselis, Johan Meivert, Armin Metzger, Peter Morgan, Ninna Morin, Ingvar Nordstrom, Chris O'Dea, Klaus Olsen, Ismo Paukamainen, Nathalie Phung, Helmut Pichler, Salvatore Reale, Stuart Reid, Hans Rombouts, Petri Säilynoja, Soile Sainio, Lars-Erik Sandberg, Dakar Shalom, Jian Shen, Marco Sogliani, Lucjan Stapp, Yaron Tsubery, Sabine Uhde, Stephanie Ulrich, Tommi Välimäki, Jurian Van de Laar, Marnix Van den Ent, António Vieira Melo, Wenye Xu, Ester Zabar, Wenqiang Zheng, Peter Zimmerer, Stevan Zivanovic, and Terry Zuo.

This document was formally approved for release by the General Assembly of the ISTQB® on May 31, 2014.

0. Introduction to this Syllabus

0.1 Purpose of this Document

This syllabus forms the basis for the International Software Testing Qualification at the Foundation Level for the Agile Tester. The ISTQB® provides this syllabus as follows:

- To National Boards, to translate into their local language and to accredit training providers. National Boards may adapt the syllabus to their particular language needs and modify the references to adapt to their local publications.
- To Exam Boards, to derive examination questions in their local language adapted to the learning objectives for each syllabus.
- To training providers, to produce courseware and determine appropriate teaching methods.
- To certification candidates, to prepare for the exam (as part of a training course or independently).
- To the international software and systems engineering community, to advance the profession of software and systems testing, and as a basis for books and articles.

The ISTQB® may allow other entities to use this syllabus for other purposes, provided they seek and obtain prior written permission.

0.2 Overview

The Foundation Level Agile Tester Overview document [ISTQB_FA_OVIEW] includes the following information:

- Business Outcomes for the syllabus
- Summary for the syllabus
- Relationships among the syllabi
- Description of cognitive levels (K-levels)
- Appendices

0.3 Examinable Learning Objectives

The Learning Objectives support the Business Outcomes and are used to create the examination for achieving the Certified Tester Foundation Level—Agile Tester Certification. In general, all parts of this syllabus are examinable at a K1 level. That is, the candidate will recognize, remember, and recall a term or concept. The specific learning objectives at K1, K2, and K3 levels are shown at the beginning of the pertinent chapter.

1. Agile Software Development - 150 mins.

Keywords

Agile Manifesto, Agile software development, incremental development model, iterative development model, software lifecycle, test automation, test basis, test-driven development, test oracle, user story

Learning Objectives for Agile Software Development

1.1 The Fundamentals of Agile Software Development

FA-1.1.1 (K1) Recall the basic concept of Agile software development based on the Agile Manifesto
FA-1.1.2 (K2) Understand the advantages of the whole-team approach
FA-1.1.3 (K2) Understand the benefits of early and frequent feedback

1.2 Aspects of Agile Approaches

FA-1.2.1 (K1) Recall Agile software development approaches
FA-1.2.2 (K3) Write testable user stories in collaboration with developers and business representatives
FA-1.2.3 (K2) Understand how retrospectives can be used as a mechanism for process improvement in Agile projects
FA-1.2.4 (K2) Understand the use and purpose of continuous integration
FA-1.2.5 (K1) Know the differences between iteration and release planning, and how a tester adds value in each of these activities

1.1 The Fundamentals of Agile Software Development

A tester on an Agile project will work differently than one working on a traditional project. Testers must understand the values and principles that underpin Agile projects, and how testers are an integral part of a whole-team approach together with developers and business representatives. The members in an Agile project communicate with each other early and frequently, which helps with removing defects early and developing a quality product.

1.1.1 Agile Software Development and the Agile Manifesto

In 2001, a group of individuals, representing the most widely used lightweight software development methodologies, agreed on a common set of values and principles which became known as the Manifesto for Agile Software Development or the Agile Manifesto [Agilemanifesto]. The Agile Manifesto contains four statements of values:

- Individuals and interactions *over* processes and tools
- Working software *over* comprehensive documentation
- Customer collaboration *over* contract negotiation
- Responding to change *over* following a plan

The Agile Manifesto argues that although the concepts on the right have value, those on the left have greater value.

Individuals and Interactions

Agile development is very people-centered. Teams of people build software, and it is through continuous communication and interaction, rather than a reliance on tools or processes, that teams can work most effectively.

Working Software

From a customer perspective, working software is much more useful and valuable than overly detailed documentation and it provides an opportunity to give the development team rapid feedback. In addition, because working software, albeit with reduced functionality, is available much earlier in the development lifecycle, Agile development can confer significant time-to-market advantage. Agile development is, therefore, especially useful in rapidly changing business environments where the problems and/or solutions are unclear or where the business wishes to innovate in new problem domains.

Customer Collaboration

Customers often find great difficulty in specifying the system that they require. Collaborating directly with the customer improves the likelihood of understanding exactly what the customer requires. While having contracts with customers may be important, working in regular and close collaboration with them is likely to bring more success to the project.

Responding to Change

Change is inevitable in software projects. The environment in which the business operates, legislation, competitor activity, technology advances, and other factors can have major influences on the project and its objectives. These factors must be accommodated by the development process. As such, having flexibility in work practices to embrace change is more important than simply adhering rigidly to a plan.

Principles

The core Agile Manifesto values are captured in twelve principles:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, at intervals of between a few weeks to a few months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

The different Agile methodologies provide prescriptive practices to put these values and principles into action.

1.1.2 Whole-Team Approach

The whole-team approach means involving everyone with the knowledge and skills necessary to ensure project success. The team includes representatives from the customer and other business stakeholders who determine product features. The team should be relatively small; successful teams have been observed with as few as three people and as many as nine. Ideally, the whole team shares the same workspace, as co-location strongly facilitates communication and interaction. The whole-team approach is supported through the daily stand-up meetings (see Section 2.2.1) involving all members of the team, where work progress is communicated and any impediments to progress are highlighted. The whole-team approach promotes more effective and efficient team dynamics.

The use of a whole-team approach to product development is one of the main benefits of Agile development. Its benefits include:

- Enhancing communication and collaboration within the team
- Enabling the various skill sets within the team to be leveraged to the benefit of the project
- Making quality everyone’s responsibility

The whole team is responsible for quality in Agile projects. The essence of the whole-team approach lies in the testers, developers, and the business representatives working together in every step of the development process. Testers will work closely with both developers and business representatives to ensure that the desired quality levels are achieved. This includes supporting and collaborating with business representatives to help them create suitable acceptance tests, working with developers to agree on the testing strategy, and deciding on test automation approaches. Testers can thus transfer and extend testing knowledge to other team members and influence the development of the product.

The whole team is involved in any consultations or meetings in which product features are presented, analyzed, or estimated. The concept of involving testers, developers, and business representatives in all feature discussions is known as the power of three [Crispin08].

1.1.3 Early and Frequent Feedback

Agile projects have short iterations enabling the project team to receive early and continuous feedback on product quality throughout the development lifecycle. One way to provide rapid feedback is by continuous integration (see Section 1.2.4).

When sequential development approaches are used, the customer often does not see the product until the project is nearly completed. At that point, it is often too late for the development team to effectively address any issues the customer may have. By getting frequent customer feedback as the project progresses, Agile teams can incorporate most new changes into the product development process. Early and frequent feedback helps the team focus on the features with the highest business value, or associated risk, and these are delivered to the customer first. It also helps manage the team better since the capability of the team is transparent to everyone. For example, how much work can we do in a sprint or iteration? What could help us go faster? What is preventing us from doing so?

The benefits of early and frequent feedback include:

- Avoiding requirements misunderstandings, which may not have been detected until later in the development cycle when they are more expensive to fix.
- Clarifying customer feature requests, making them available for customer use early. This way, the product better reflects what the customer wants.
- Discovering (via continuous integration), isolating, and resolving quality problems early.
- Providing information to the Agile team regarding its productivity and ability to deliver.
- Promoting consistent project momentum.

1.2 Aspects of Agile Approaches

There are a number of Agile approaches in use by organizations. Common practices across most Agile organizations include collaborative user story creation, retrospectives, continuous integration, and planning for each iteration as well as for overall release. This subsection describes some of the Agile approaches.

1.2.1 Agile Software Development Approaches

There are several Agile approaches, each of which implements the values and principles of the Agile Manifesto in different ways. In this syllabus, three representatives of Agile approaches are considered: Extreme Programming (XP), Scrum, and Kanban.

Extreme Programming
Extreme Programming (XP), originally introduced by Kent Beck [Beck04], is an Agile approach to software development described by certain values, principles, and development practices.

XP embraces five values to guide development: communication, simplicity, feedback, courage, and respect.

XP describes a set of principles as additional guidelines: humanity, economics, mutual benefit, self-similarity, improvement, diversity, reflection, flow, opportunity, redundancy, failure, quality, baby steps, and accepted responsibility.

XP describes thirteen primary practices: sit together, whole team, informative workspace, energized work, pair programming, stories, weekly cycle, quarterly cycle, slack, ten-minute build, continuous integration, test first programming, and incremental design.

Many of the Agile software development approaches in use today are influenced by XP and its values and principles. For example, Agile teams following Scrum often incorporate XP practices.

Scrum
Scrum is an Agile management framework which contains the following constituent instruments and practices [Schwaber01]:

- Sprint: Scrum divides a project into iterations (called sprints) of fixed length (usually two to four weeks).
- Product Increment: Each sprint results in a potentially releasable/shippable product (called an increment).
- Product Backlog: The product owner manages a prioritized list of planned product items (called the product backlog). The product backlog evolves from sprint to sprint (called backlog refinement).
- Sprint Backlog: At the start of each sprint, the Scrum team selects a set of highest priority items (called the sprint backlog) from the product backlog. Since the Scrum team, not the product owner, selects the items to be realized within the sprint, the selection is referred to as being on the pull principle rather than the push principle.
- Definition of Done: To make sure that there is a potentially releasable product at each sprint's end, the Scrum team discusses and defines appropriate criteria for sprint completion. The discussion deepens the team's understanding of the backlog items and the product requirements.
- Timeboxing: Only those tasks, requirements, or features that the team expects to finish within the sprint are part of the sprint backlog. If the development team cannot finish a task within a sprint, the associated product features are removed from the sprint and the task is moved back into the product backlog. Timeboxing applies not only to tasks, but in other situations (e.g., enforcing meeting start and end times).
- Transparency: The development team reports and updates sprint status on a daily basis at a meeting called the daily scrum. This makes the content and progress of the current sprint, including test results, visible to the team, management, and all interested parties. For example, the development team can show sprint status on a whiteboard.

Scrum defines three roles:

- Scrum Master: ensures that Scrum practices and rules are implemented and followed, and resolves any violations, resource issues, or other impediments that could prevent the team from following the practices and rules. This person is not the team lead, but a coach.
- Product Owner: represents the customer, and generates, maintains, and prioritizes the product backlog. This person is not the team lead.
- Development Team: develop and test the product. The team is self-organized: There is no team lead, so the team makes the decisions. The team is also cross-functional (see Section 2.3.2 and Section 3.1.4).

Scrum (as opposed to XP) does not dictate specific software development techniques (e.g., test first programming). In addition, Scrum does not provide guidance on how testing has to be done in a Scrum project.

Kanban
Kanban [Anderson13] is a management approach that is sometimes used in Agile projects. The general objective is to visualize and optimize the flow of work within a value-added chain. Kanban utilizes three instruments [Linz14]:

- Kanban Board: The value chain to be managed is visualized by a Kanban board. Each column shows a station, which is a set of related activities, e.g., development or testing. The items to be produced or tasks to be processed are symbolized by tickets moving from left to right across the board through the stations.
- Work-in-Progress Limit: The amount of parallel active tasks is strictly limited. This is controlled by the maximum number of tickets allowed for a station and/or globally for the board. Whenever a station has free capacity, the worker pulls a ticket from the predecessor station.
- Lead Time: Kanban is used to optimize the continuous flow of tasks by minimizing the (average) lead time for the complete value stream.

Kanban features some similarities to Scrum. In both frameworks, visualizing the active tasks (e.g., on a public whiteboard) provides transparency of content and progress of tasks. Tasks not yet scheduled are waiting in a backlog and moved onto the Kanban board as soon as there is new space (production capacity) available.

Iterations or sprints are optional in Kanban. The Kanban process allows releasing its deliverables item by item, rather than as part of a release. Timeboxing as a synchronizing mechanism, therefore, is optional, unlike in Scrum, which synchronizes all tasks within a sprint.

1.2.2 Collaborative User Story Creation

Poor specifications are often a major reason for project failure. Specification problems can result from the users' lack of insight into their true needs, absence of a global vision for the system, redundant or contradictory features, and other miscommunications. In Agile development, user stories are written to capture requirements from the perspectives of developers, testers, and business representatives. In sequential development, this shared vision of a feature is accomplished through formal reviews after requirements are written; in Agile development, this shared vision is accomplished through frequent informal reviews while the requirements are being written.

The user stories must address both functional and non-functional characteristics. Each story includes acceptance criteria for these characteristics. These criteria should be defined in collaboration between business representatives, developers, and testers. They provide developers and testers with an extended vision of the feature that business representatives will validate. An Agile team considers a task finished when a set of acceptance criteria have been satisfied.

Typically, the tester's unique perspective will improve the user story by identifying missing details or non-functional requirements. A tester can contribute by asking business representatives open-ended questions about the user story, proposing ways to test the user story, and confirming the acceptance criteria.

The collaborative authorship of the user story can use techniques such as brainstorming and mind mapping. The tester may use the INVEST technique [INVEST]:

- Independent
- Negotiable
- Valuable
- Estimable
- Small
- Testable

According to the 3C concept [Jeffries00], a user story is the conjunction of three elements:

- Card: The card is the physical media describing a user story. It identifies the requirement, its criticality, expected development and test duration, and the acceptance criteria for that story. The description has to be accurate, as it will be used in the product backlog.

- Conversation: The conversation explains how the software will be used. The conversation can be documented or verbal. Testers, having a different point of view than developers and business representatives [ISTQB_FL_SYL], bring valuable input to the exchange of thoughts, opinions, and experiences. Conversation begins during the release-planning phase and continues when the story is scheduled.
- Confirmation: The acceptance criteria, discussed in the conversation, are used to confirm that the story is done. These acceptance criteria may span multiple user stories. Both positive and negative tests should be used to cover the criteria. During confirmation, various participants play the role of a tester. These can include developers as well as specialists focused on performance, security, interoperability, and other quality characteristics. To confirm a story as done, the defined acceptance criteria should be tested and shown to be satisfied.

Agile teams vary in terms of how they document user stories. Regardless of the approach taken to document user stories, documentation should be concise, sufficient, and necessary.

1.2.3 Retrospectives

In Agile development, a retrospective is a meeting held at the end of each iteration to discuss what was successful, what could be improved, and how to incorporate the improvements and retain the successes in future iterations. Retrospectives cover topics such as the process, people, organizations, relationships, and tools. Regularly conducted retrospective meetings, when appropriate follow up activities occur, are critical to self-organization and continual improvement of development and testing.

Retrospectives can result in test-related improvement decisions focused on test effectiveness, test productivity, test case quality, and team satisfaction. They may also address the testability of the applications, user stories, features, or system interfaces. Root cause analysis of defects can drive testing and development improvements. In general, teams should implement only a few improvements per iteration. This allows for continuous improvement at a sustained pace.

The timing and organization of the retrospective depends on the particular Agile method followed. Business representatives and the team attend each retrospective as participants while the facilitator organizes and runs the meeting. In some cases, the teams may invite other participants to the meeting.

Testers should play an important role in the retrospectives. Testers are part of the team and bring their unique perspective [ISTQB_FL_SYL], Section 1.5. Testing occurs in each sprint and vitally contributes to success. All team members, testers and non-testers, can provide input on both testing and non-testing activities.

Retrospectives must occur within a professional environment characterized by mutual trust. The attributes of a successful retrospective are the same as those for any other review as is discussed in the Foundation Level syllabus [ISTQB_FL_SYL], Section 3.2.

1.2.4 Continuous Integration

Delivery of a product increment requires reliable, working, integrated software at the end of every sprint. Continuous integration addresses this challenge by merging all changes made to the software and integrating all changed components regularly, at least once a day. Configuration management, compilation, software build, deployment, and testing are wrapped into a single, automated, repeatable process. Since developers integrate their work constantly, build constantly, and test constantly, defects in code are detected more quickly.

Following the developers' coding, debugging, and check-in of code into a shared source code repository, a continuous integration process consists of the following automated activities:

- Static code analysis: executing static code analysis and reporting results
- Compile: compiling and linking the code, generating the executable files
- Unit test: executing the unit tests, checking code coverage and reporting test results
- Deploy: installing the build into a test environment
- Integration test: executing the integration tests and reporting results
- Report (dashboard): posting the status of all these activities to a publicly visible location or e-mailing status to the team

An automated build and test process takes place on a daily basis and detects integration errors early and quickly. Continuous integration allows Agile testers to run automated tests regularly, in some cases as part of the continuous integration process itself, and send quick feedback to the team on the quality of the code. These test results are visible to all team members, especially when automated reports are integrated into the process. Automated regression testing can be continuous throughout the iteration. Good automated regression tests cover as much functionality as possible, including user stories delivered in the previous iterations. Good coverage in the automated regression tests helps support building (and testing) large integrated systems. When the regression testing is automated, the Agile testers are freed to concentrate their manual testing on new features, implemented changes, and confirmation testing of defect fixes.

In addition to automated tests, organizations using continuous integration typically use build tools to implement continuous quality control. In addition to running unit and integration tests, such tools can run additional static and dynamic tests, measure and profile performance, extract and format documentation from the source code, and facilitate manual quality assurance processes. This continuous application of quality control aims to improve the quality of the product as well as reduce the time taken to deliver it by replacing the traditional practice of applying quality control after completing all development.

Build tools can be linked to automatic deployment tools, which can fetch the appropriate build from the continuous integration or build server and deploy it into one or more development, test, staging, or even production environments. This reduces the errors and delays associated with relying on specialized staff or programmers to install releases in these environments.

Continuous integration can provide the following benefits:
- Allows earlier detection and easier root cause analysis of integration problems and conflicting changes
- Gives the development team regular feedback on whether the code is working
- Keeps the version of the software being tested within a day of the version being developed
- Reduces regression risk associated with developer code refactoring due to rapid re-testing of the code base after each small set of changes
- Provides confidence that each day's development work is based on a solid foundation
- Makes progress toward the completion of the product increment visible, encouraging developers and testers
- Eliminates the schedule risks associated with big-bang integration
- Provides constant availability of executable software throughout the sprint for testing, demonstration, or education purposes
- Reduces repetitive manual testing activities
- Provides quick feedback on decisions made to improve quality and tests

However, continuous integration is not without its risks and challenges:
- Continuous integration tools have to be introduced and maintained
- The continuous integration process must be defined and established
- Test automation requires additional resources and can be complex to establish
- Thorough test coverage is essential to achieve automated testing advantages

- Teams sometimes over-rely on unit tests and perform too little system and acceptance testing

Continuous integration requires the use of tools, including tools for testing, tools for automating the build process, and tools for version control.

1.2.5 Release and Iteration Planning

As mentioned in the Foundation Level syllabus [ISTQB_FL_SYL], planning is an on-going activity, and this is the case in Agile lifecycles as well. For Agile lifecycles, two kinds of planning occur, release planning and iteration planning.

Release planning looks ahead to the release of a product, often a few months ahead of the start of a project. Release planning defines and re-defines the product backlog, and may involve refining larger user stories into a collection of smaller stories. Release planning provides the basis for a test approach and test plan spanning all iterations. Release plans are high-level.

In release planning, business representatives establish and prioritize the user stories for the release, in collaboration with the team (see Section 1.2.2). Based on these user stories, project and quality risks are identified and a high-level effort estimation is performed (see Section 3.2).

Testers are involved in release planning and especially add value in the following activities:
- Defining testable user stories, including acceptance criteria
- Participating in project and quality risk analyses
- Estimating testing effort associated with the user stories
- Defining the necessary test levels
- Planning the testing for the release

After release planning is done, iteration planning for the first iteration starts. Iteration planning looks ahead to the end of a single iteration and is concerned with the iteration backlog.

In iteration planning, the team selects user stories from the prioritized release backlog, elaborates the user stories, performs a risk analysis for the user stories, and estimates the work needed for each user story. If a user story is too vague and attempts to clarify it have failed, the team can refuse to accept it and use the next user story based on priority. The business representatives must answer the team's questions about each story so the team can understand what they should implement and how to test each story.

The number of stories selected is based on established team velocity and the estimated size of the selected user stories. After the contents of the iteration are finalized, the user stories are broken into tasks, which will be carried out by the appropriate team members.

Testers are involved in iteration planning and especially add value in the following activities:
- Participating in the detailed risk analysis of user stories
- Determining the testability of the user stories
- Creating acceptance tests for the user stories
- Breaking down user stories into tasks (particularly testing tasks)
- Estimating testing effort for all testing tasks
- Identifying functional and non-functional aspects of the system to be tested
- Supporting and participating in test automation at multiple levels of testing

Release plans may change as the project proceeds, including changes to individual user stories in the product backlog. These changes may be triggered by internal or external factors. Internal factors include delivery capabilities, velocity, and technical issues. External factors include the discovery of

new markets and opportunities, new competitors, or business threats that may change release objectives and/or target dates. In addition, iteration plans may change during an iteration. For example, a particular user story that was considered relatively simple during estimation might prove more complex than expected.

These changes can be challenging for testers. Testers must understand the big picture of the release for test planning purposes, and they must have an adequate test basis and test oracle in each iteration for test development purposes as discussed in the Foundation Level syllabus [ISTQB_FL_SYL], Section 1.4. The required information must be available to the tester early, and yet change must be embraced according to Agile principles. This dilemma requires careful decisions about test strategies and test documentation. For more on Agile testing challenges, see [Black09], Chapter 12.

Release and iteration planning should address test planning as well as planning for development activities. Particular test-related issues to address include:

- The scope of testing, the extent of testing for those areas in scope, the test goals, and the reasons for these decisions.
- The team members who will carry out the test activities.
- The test environment and test data needed, when they are needed, and whether any additions or changes to the test environment and/or data will occur prior to or during the project.
- The timing, sequencing, dependencies, and prerequisites for the functional and non-functional test activities (e.g., how frequently to run regression tests, which features depend on other features or test data, etc.), including how the test activities relate to and depend on development activities.
- The project and quality risks to be addressed (see Section 3.2.1).

In addition, the larger team estimation effort should include consideration of the time and effort needed to complete the required testing activities.

2. Fundamental Agile Testing Principles, Practices, and Processes – 105 mins.

Keywords
build verification test, configuration item, configuration management

Learning Objectives for Fundamental Agile Testing Principles, Practices, and Processes

2.1 The Differences between Testing in Traditional and Agile Approaches

FA-2.1.1 (K2) Describe the differences between testing activities in Agile projects and non-Agile projects
FA-2.1.2 (K2) Describe how development and testing activities are integrated in Agile projects
FA-2.1.3 (K2) Describe the role of independent testing in Agile projects

2.2 Status of Testing in Agile Projects

FA-2.2.1 (K2) Describe the tools and techniques used to communicate the status of testing in an Agile project, including test progress and product quality
FA-2.2.2 (K2) Describe the process of evolving tests across multiple iterations and explain why test automation is important to manage regression risk in Agile projects

2.3 Role and Skills of a Tester in an Agile Team

FA-2.3.1 (K2) Understand the skills (people, domain, and testing) of a tester in an Agile team
FA-2.3.2 (K2) Understand the role of a tester within an Agile team

2.1 The Differences between Testing in Traditional and Agile Approaches

As described in the Foundation Level syllabus [ISTQB_FL_SYL] and in [Black09], test activities are related to development activities, and thus testing varies in different lifecycles. Testers must understand the differences between testing in traditional lifecycle models (e.g., sequential such as the V-model or iterative such as RUP) and Agile lifecycles in order to work effectively and efficiently. The Agile models differ in terms of the way testing and development activities are integrated, the project work products, the names, entry and exit criteria used for various levels of testing, the use of tools, and how independent testing can be effectively utilized.

Testers should remember that organizations vary considerably in their implementation of lifecycles. Deviation from the ideals of Agile lifecycles (see Section 1.1) may represent intelligent customization and adaptation of the practices. The ability to adapt to the context of a given project, including the software development practices actually followed, is a key success factor for testers.

2.1.1 Testing and Development Activities

One of the main differences between traditional lifecycles and Agile lifecycles is the idea of very short iterations, each iteration resulting in working software that delivers features of value to business stakeholders. At the beginning of the project, there is a release planning period. This is followed by a sequence of iterations. At the beginning of each iteration, there is an iteration planning period. Once iteration scope is established, the selected user stories are developed, integrated with the system, and tested. These iterations are highly dynamic, with development, integration, and testing activities taking place throughout each iteration, and with considerable parallelism and overlap. Testing activities occur throughout the iteration, not as a final activity.

Testers, developers, and business stakeholders all have a role in testing, as with traditional lifecycles. Developers perform unit tests as they develop features from the user stories. Testers then test those features. Business stakeholders also test the stories during implementation. Business stakeholders might use written test cases, but they also might simply experiment with and use the feature in order to provide fast feedback to the development team.

In some cases, hardening or stabilization iterations occur periodically to resolve any lingering defects and other forms of technical debt. However, the best practice is that no feature is considered done until it has been integrated and tested with the system [Goucher09]. Another good practice is to address defects remaining from the previous iteration at the beginning of the next iteration, as part of the backlog for that iteration (referred to as “fix bugs first”). However, some complain that this practice results in a situation where the total work to be done in the iteration is unknown and it will be more difficult to estimate when the remaining features can be done. At the end of the sequence of iterations, there can be a set of release activities to get the software ready for delivery, though in some cases delivery occurs at the end of each iteration.

When risk-based testing is used as one of the test strategies, a high-level risk analysis occurs during release planning, with testers often driving that analysis. However, the specific quality risks associated with each iteration are identified and assessed in iteration planning. This risk analysis can influence the sequence of development as well as the priority and depth of testing for the features. It also influences the estimation of the test effort required for each feature (see Section 3.2).

In some Agile practices (e.g., Extreme Programming), pairing is used. Pairing can involve testers working together in twos to test a feature. Pairing can also involve a tester working collaboratively with a developer to develop and test a feature. Pairing can be difficult when the test team is distributed, but processes and tools can help enable distributed pairing. For more information on distributed work, see [ISTQB_ALTM_SYL], Section 2.8.

Testers may also serve as testing and quality coaches within the team, sharing testing knowledge and supporting quality assurance work within the team. This promotes a sense of collective ownership of quality of the product.

Test automation at all levels of testing occurs in many Agile teams, and this can mean that testers spend time creating, executing, monitoring, and maintaining automated tests and results. Because of the heavy use of test automation, a higher percentage of the manual testing on Agile projects tends to be done using experience-based and defect-based techniques such as software attacks, exploratory testing, and error guessing (see [ISTQB_ALTA_SYL], Sections 3.3 and 3.4 and [ISTQB_FL_SYL], Section 4.5). While developers will focus on creating unit tests, testers should focus on creating automated integration, system, and system integration tests. This leads to a tendency for Agile teams to favor testers with a strong technical and test automation background.

One core Agile principle is that change may occur throughout the project. Therefore, lightweight work product documentation is favored in Agile projects. Changes to existing features have testing implications, especially regression testing implications. The use of automated testing is one way of managing the amount of test effort associated with change. However, it's important that the rate of change not exceed the project team's ability to deal with the risks associated with those changes.

2.1.2 Project Work Products

Project work products of immediate interest to Agile testers typically fall into three categories:

1. Business-oriented work products that describe what is needed (e.g., requirements specifications) and how to use it (e.g., user documentation)
2. Development work products that describe how the system is built (e.g., database entity-relationship diagrams), that actually implement the system (e.g., code), or that evaluate individual pieces of code (e.g., automated unit tests)
3. Test work products that describe how the system is tested (e.g., test strategies and plans), that actually test the system (e.g., manual and automated tests), or that present test results (e.g., test dashboards as discussed in Section 2.2.1)

In a typical Agile project, it is a common practice to avoid producing vast amounts of documentation. Instead, focus is more on having working software, together with automated tests that demonstrate conformance to requirements. This encouragement to reduce documentation applies only to documentation that does not deliver value to the customer. In a successful Agile project, a balance is struck between increasing efficiency by reducing documentation and providing sufficient documentation to support business, testing, development, and maintenance activities. The team must make a decision during release planning about which work products are required and what level of work product documentation is needed.

Typical business-oriented work products on Agile projects include user stories and acceptance criteria. User stories are the Agile form of requirements specifications, and should explain how the system should behave with respect to a single, coherent feature or function. A user story should define a feature small enough to be completed in a single iteration. Larger collections of related features, or a collection of sub-features that make up a single complex feature, may be referred to as "epics". Epics may include user stories for different development teams. For example, one user story can describe what is required at the API-level (middleware) while another story describes what is needed at the UI-level (application). These collections may be developed over a series of sprints. Each epic and its user stories should have associated acceptance criteria.

Typical developer work products on Agile projects include code. Agile developers also often create automated unit tests. These tests might be created after the development of code. In some cases, though, developers create tests incrementally, before each portion of the code is written, in order to provide a way of verifying, once that portion of code is written, whether it works as expected. While

this approach is referred to as test first or test-driven development, in reality the tests are more a form of executable low-level design specifications rather than tests [Beck02].

Typical tester work products on Agile projects include automated tests, as well as documents such as test plans, quality risk catalogs, manual tests, defect reports, and test results logs. The documents are captured in as lightweight a fashion as possible, which is often also true of these documents in traditional lifecycles. Testers will also produce test metrics from defect reports and test results logs, and again there is an emphasis on a lightweight approach.

In some Agile implementations, especially regulated, safety critical, distributed, or highly complex projects and products, further formalization of these work products is required. For example, some teams transform user stories and acceptance criteria into more formal requirements specifications. Vertical and horizontal traceability reports may be prepared to satisfy auditors, regulations, and other requirements.

2.1.3 Test Levels

Test levels are test activities that are logically related, often by the maturity or completeness of the item under test.

In sequential lifecycle models, the test levels are often defined such that the exit criteria of one level are part of the entry criteria for the next level. In some iterative models, this rule does not apply. Test levels overlap. Requirement specification, design specification, and development activities may overlap with test levels.

In some Agile lifecycles, overlap occurs because changes to requirements, design, and code can happen at any point in an iteration. While Scrum, in theory, does not allow changes to the user stories after iteration planning, in practice such changes sometimes occur. During an iteration, any given user story will typically progress sequentially through the following test activities:

- Unit testing, typically done by the developer
- Feature acceptance testing, which is sometimes broken into two activities:
    - Feature verification testing, which is often automated, may be done by developers or testers, and involves testing against the user story's acceptance criteria
    - Feature validation testing, which is usually manual and can involve developers, testers, and business stakeholders working collaboratively to determine whether the feature is fit for use, to improve visibility of the progress made, and to receive real feedback from the business stakeholders

In addition, there is often a parallel process of regression testing occurring throughout the iteration. This involves re-running the automated unit tests and feature verification tests from the current iteration and previous iterations, usually via a continuous integration framework.

In some Agile projects, there may be a system test level, which starts once the first user story is ready for such testing. This can involve executing functional tests, as well as non-functional tests for performance, reliability, usability, and other relevant test types.

Agile teams can employ various forms of acceptance testing (using the term as explained in the Foundation Level syllabus [ISTQB_FL_SYL]). Internal alpha tests and external beta tests may occur, either at the close of each iteration, after the completion of each iteration, or after a series of iterations. User acceptance tests, operational acceptance tests, regulatory acceptance tests, and contract acceptance tests also may occur, either at the close of each iteration, after the completion of each iteration, or after a series of iterations.

2.1.4 Testing and Configuration Management

Agile projects often involve heavy use of automated tools to develop, test, and manage software development. Developers use tools for static analysis, unit testing, and code coverage. Developers continuously check the code and unit tests into a configuration management system, using automated build and test frameworks. These frameworks allow the continuous integration of new software with the system, with the static analysis and unit tests run repeatedly as new software is checked in [Kubaczkowski].

These automated tests can also include functional tests at the integration and system levels. Such functional automated tests may be created using functional testing harnesses, open-source user interface functional test tools, or commercial tools, and can be integrated with the automated tests run as part of the continuous integration framework. In some cases, due to the duration of the functional tests, the functional tests are separated from the unit tests and run less frequently. For example, unit tests may be run each time new software is checked in, while the longer functional tests are run only every few days.

One goal of the automated tests is to confirm that the build is functioning and installable. If any automated test fails, the team should fix the underlying defect in time for the next code check-in. This requires an investment in real-time test reporting to provide good visibility into test results. This approach helps reduce expensive and inefficient cycles of "build-install-fail-rebuild-reinstall" that can occur in many traditional projects, since changes that break the build or cause software to fail to install are detected quickly.

Automated testing and build tools help to manage the regression risk associated with the frequent change that often occurs in Agile projects. However, over-reliance on automated unit testing alone to manage these risks can be a problem, as unit testing often has limited defect detection effectiveness [Jones11]. Automated tests at the integration and system levels are also required.

2.1.5 Organizational Options for Independent Testing

As discussed in the Foundation Level syllabus [ISTQB_FL_SYL], independent testers are often more effective at finding defects. In some Agile teams, developers create many of the tests in the form of automated tests. One or more testers may be embedded within the team, performing many of the testing tasks. However, given those testers' position within the team, there is a risk of loss of independence and objective evaluation.

Other Agile teams retain fully independent, separate test teams, and assign testers on-demand during the final days of each sprint. This can preserve independence, and these testers can provide an objective, unbiased evaluation of the software. However, time pressures, lack of understanding of the new features in the product, and relationship issues with business stakeholders and developers often lead to problems with this approach.

A third option is to have an independent, separate test team where testers are assigned to Agile teams on a long-term basis, at the beginning of the project, allowing them to maintain their independence while gaining a good understanding of the product and strong relationships with other team members. In addition, the independent test team can have specialized testers outside of the Agile teams to work on long-term and/or iteration-independent activities, such as developing automated test tools, carrying out non-functional testing, creating and supporting test environments and data, and carrying out test levels that might not fit well within a sprint (e.g., system integration testing).

2.2 Status of Testing in Agile Projects

Change takes place rapidly in Agile projects. This change means that test status, test progress, and product quality constantly evolve, and testers must devise ways to get that information to the team so that they can make decisions to stay on track for successful completion of each iteration. In addition, change can affect existing features from previous iterations. Therefore, manual and automated tests must be updated to deal effectively with regression risk.

2.2.1 Communicating Test Status, Progress, and Product Quality

Agile teams progress by having working software at the end of each iteration. To determine when the team will have working software, they need to monitor the progress of all work items in the iteration and release. Testers in Agile teams utilize various methods to record test progress and status, including test automation results, progression of test tasks and stories on the Agile task board, and burndown charts showing the team's progress. These can then be communicated to the rest of the team using media such as wiki dashboards and dashboard-style emails, as well as verbally during stand-up meetings. Agile teams may use tools that automatically generate status reports based on test results and task progress, which in turn update wiki-style dashboards and emails. This method of communication also gathers metrics from the testing process, which can be used in process improvement. Communicating test status in such an automated manner also frees testers' time to focus on designing and executing more test cases.

Teams may use burndown charts to track progress across the entire release and within each iteration. A burndown chart [Crispin08] represents the amount of work left to be done against time allocated to the release or iteration.

To provide an instant, detailed visual representation of the whole team's current status, including the status of testing, teams may use Agile task boards. The story cards, development tasks, test tasks, and other tasks created during iteration planning (see Section 1.2.5) are captured on the task board, often using color-coordinated cards to determine the task type. During the iteration, progress is managed via the movement of these tasks across the task board into columns such as *to do*, *work in progress*, *verify*, and *done*. Agile teams may use tools to maintain their story cards and Agile task boards, which can automate dashboards and status updates.

Testing tasks on the task board relate to the acceptance criteria defined for the user stories. As test automation scripts, manual tests, and exploratory tests for a test task achieve a passing status, the task moves into the *done* column of the task board. The whole team reviews the status of the task board regularly, often during the daily stand-up meetings, to ensure tasks are moving across the board at an acceptable rate. If any tasks (including testing tasks) are not moving or are moving too slowly, the team reviews and addresses any issues that may be blocking the progress of those tasks.

The daily stand-up meeting includes all members of the Agile team including testers. At this meeting, they communicate their current status. The agenda for each member is [Agile Alliance Guide]:

- What have you completed since the last meeting?
- What do you plan to complete by the next meeting?
- What is getting in your way?

Any issues that may block test progress are communicated during the daily stand-up meetings, so the whole team is aware of the issues and can resolve them accordingly.

To improve the overall product quality, many Agile teams perform customer satisfaction surveys to receive feedback on whether the product meets customer expectations. Teams may use other metrics similar to those captured in traditional development methodologies, such as test pass/fail rates, defect discovery rates, confirmation and regression test results, defect density, defects found and fixed, requirements coverage, risk coverage, code coverage, and code churn to improve the product quality.

As with any lifecycle, the metrics captured and reported should be relevant and aid decision-making. Metrics should not be used to reward, punish, or isolate any team members.

2.2.2 Managing Regression Risk with Evolving Manual and Automated Test Cases

In an Agile project, as each iteration completes, the product grows. Therefore, the scope of testing also increases. Along with testing the code changes made in the current iteration, testers also need to verify no regression has been introduced on features that were developed and tested in previous iterations. The risk of introducing regression in Agile development is high due to extensive code churn (lines of code added, modified, or deleted from one version to another). Since responding to change is a key Agile principle, changes can also be made to previously delivered features to meet business needs. In order to maintain velocity without incurring a large amount of technical debt, it is critical that teams invest in test automation at all test levels as early as possible. It is also critical that all test assets such as automated tests, manual test cases, test data, and other testing artifacts are kept up-to-date with each iteration. It is highly recommended that all test assets be maintained in a configuration management tool in order to enable version control, to ensure ease of access by all team members, and to support making changes as required due to changing functionality while still preserving the historic information of the test assets.

Because complete repetition of all tests is seldom possible, especially in tight-timeline Agile projects, testers need to allocate time in each iteration to review manual and automated test cases from previous and current iterations to select test cases that may be candidates for the regression test suite, and to retire test cases that are no longer relevant. Tests written in earlier iterations to verify specific features may have little value in later iterations due to feature changes or new features which alter the way those earlier features behave.

While reviewing test cases, testers should consider suitability for automation. The team needs to automate as many tests as possible from previous and current iterations. This allows automated regression tests to reduce regression risk with less effort than manual regression testing would require. This reduced regression test effort frees the testers to more thoroughly test new features and functions in the current iteration.

It is critical that testers have the ability to quickly identify and update test cases from previous iterations and/or releases that are affected by the changes made in the current iteration. Defining how the team designs, writes, and stores test cases should occur during release planning. Good practices for test design and implementation need to be adopted early and applied consistently. The shorter timeframes for testing and the constant change in each iteration will increase the impact of poor test design and implementation practices.

Use of test automation, at all test levels, allows Agile teams to provide rapid feedback on product quality. Well-written automated tests provide a living document of system functionality [Crispin08]. By checking the automated tests and their corresponding test results into the configuration management system, aligned with the versioning of the product builds, Agile teams can review the functionality tested and the test results for any given build at any given point in time.

Automated unit tests are run before source code is checked into the mainline of the configuration management system to ensure the code changes do not break the software build. To reduce build breaks, which can slow down the progress of the whole team, code should not be checked in unless all automated unit tests pass. Automated unit test results provide immediate feedback on code and build quality, but not on product quality.

Automated acceptance tests are run regularly as part of the continuous integration full system build. These tests are run against a complete system build at least daily, but are generally not run with each code check-in as they take longer to run than automated unit tests and could slow down code check-

ins. The test results from automated acceptance tests provide feedback on product quality with respect to regression since the last build, but they do not provide status of overall product quality.

Automated tests can be run continuously against the system. An initial subset of automated tests to cover critical system functionality and integration points should be created immediately after a new build is deployed into the test environment. These tests are commonly known as build verification tests. Results from the build verification tests will provide instant feedback on the software after deployment, so teams don't waste time testing an unstable build.

Automated tests contained in the regression test set are generally run as part of the daily main build in the continuous integration environment, and again when a new build is deployed into the test environment. As soon as an automated regression test fails, the team stops and investigates the reasons for the failing test. The test may have failed due to legitimate functional changes in the current iteration, in which case the test and/or user story may need to be updated to reflect the new acceptance criteria. Alternatively, the test may need to be retired if another test has been built to cover the changes. However, if the test failed due to a defect, it is a good practice for the team to fix the defect prior to progressing with new features.

In addition to test automation, the following testing tasks may also be automated:

- Test data generation
- Loading test data into systems
- Deployment of builds into the test environments
- Restoration of a test environment (e.g., the database or website data files) to a baseline
- Comparison of data outputs

Automation of these tasks reduces the overhead and allows the team to spend time developing and testing new features.

2.3 Role and Skills of a Tester in an Agile Team

In an Agile team, testers must closely collaborate with all other team members and with business stakeholders. This has a number of implications in terms of the skills a tester must have and the activities they perform within an Agile team.

2.3.1 Agile Tester Skills

Agile testers should have all the skills mentioned in the Foundation Level syllabus [ISTQB_FL_SYL]. In addition to these skills, a tester in an Agile team should be competent in test automation, test-driven development, acceptance test-driven development, white-box, black-box, and experience-based testing.

As Agile methodologies depend heavily on collaboration, communication, and interaction between the team members as well as stakeholders outside the team, testers in an Agile team should have good interpersonal skills. Testers in Agile teams should:

- Be positive and solution-oriented with team members and stakeholders
- Display critical, quality-oriented, skeptical thinking about the product
- Actively acquire information from stakeholders (rather than relying entirely on written specifications)
- Accurately evaluate and report test results, test progress, and product quality
- Work effectively to define testable user stories, especially acceptance criteria, with customer representatives and stakeholders
- Collaborate within the team, working in pairs with programmers and other team members
- Respond to change quickly, including changing, adding, or improving test cases
- Plan and organize their own work

Continuous skills growth, including interpersonal skills growth, is essential for all testers, including those on Agile teams.

2.3.2 The Role of a Tester in an Agile Team

The role of a tester in an Agile team includes activities that generate and provide feedback not only on test status, test progress, and product quality, but also on process quality. In addition to the activities described elsewhere in this syllabus, these activities include:

- Understanding, implementing, and updating the test strategy
- Measuring and reporting test coverage across all applicable coverage dimensions
- Ensuring proper use of testing tools
- Configuring, using, and managing test environments and test data
- Reporting defects and working with the team to resolve them
- Coaching other team members in relevant aspects of testing
- Ensuring the appropriate testing tasks are scheduled during release and iteration planning
- Actively collaborating with developers and business stakeholders to clarify requirements, especially in terms of testability, consistency, and completeness
- Participating proactively in team retrospectives, suggesting and implementing improvements

Within an Agile team, each team member is responsible for product quality and plays a role in performing test-related tasks.

Agile organizations may encounter some test-related organizational risks:

- Testers work so closely to developers that they lose the appropriate tester mindset
- Testers become tolerant of or silent about inefficient, ineffective, or low-quality practices within the team
- Testers cannot keep pace with the incoming changes in time-constrained iterations

To mitigate these risks, organizations may consider variations for preserving independence discussed in Section 2.1.5.

3. Agile Testing Methods, Techniques, and Tools – 480 mins.

Keywords

acceptance criteria, exploratory testing, performance testing, product risk, quality risk, regression testing, test approach, test charter, test estimation, test execution automation, test strategy, test-driven development, unit test framework

Learning Objectives for Agile Testing Methods, Techniques, and Tools

3.1 Agile Testing Methods

FA-3.1.1 (K1) Recall the concepts of test-driven development, acceptance test-driven development, and behavior-driven development

FA-3.1.2 (K1) Recall the concepts of the test pyramid

FA-3.1.3 (K2) Summarize the testing quadrants and their relationships with testing levels and testing types

FA-3.1.4 (K3) For a given Agile project, practice the role of a tester in a Scrum team

3.2 Assessing Quality Risks and Estimating Test Effort

FA-3.2.1 (K3) Assess quality risks within an Agile project

FA-3.2.2 (K3) Estimate testing effort based on iteration content and quality risks

3.3 Techniques in Agile Projects

FA-3.3.1 (K3) Interpret relevant information to support testing activities

FA-3.3.2 (K2) Explain to business stakeholders how to define testable acceptance criteria

FA-3.3.3 (K3) Given a user story, write acceptance test-driven development test cases

FA-3.3.4 (K3) For both functional and non-functional behavior, write test cases using black box test design techniques based on given user stories

FA-3.3.5 (K3) Perform exploratory testing to support the testing of an Agile project

3.4 Tools in Agile Projects

FA-3.4.1 (K1) Recall different tools available to testers according to their purpose and to activities in Agile projects

3.1 Agile Testing Methods

There are certain testing practices that can be followed in every development project (agile or not) to produce quality products. These include writing tests in advance to express proper behavior, focusing on early defect prevention, detection, and removal, and ensuring that the right test types are run at the right time and as part of the right test level. Agile practitioners aim to introduce these practices early. Testers in Agile projects play a key role in guiding the use of these testing practices throughout the lifecycle.

3.1.1 Test-Driven Development, Acceptance Test-Driven Development, and Behavior-Driven Development

Test-driven development, acceptance test-driven development, and behavior-driven development are three complementary techniques in use among Agile teams to carry out testing across the various test levels. Each technique is an example of a fundamental principle of testing, the benefit of early testing and QA activities, since the tests are defined before the code is written.

Test-Driven Development
Test-driven development (TDD) is used to develop code guided by automated test cases. The process for test-driven development is:

- Add a test that captures the programmer's concept of the desired functioning of a small piece of code
- Run the test, which should fail since the code doesn't exist
- Write the code and run the test in a tight loop until the test passes
- Refactor the code after the test is passed, re-running the test to ensure it continues to pass against the refactored code
- Repeat this process for the next small piece of code, running the previous tests as well as the added tests

The tests written are primarily unit level and are code-focused, though tests may also be written at the integration or system levels. Test-driven development gained its popularity through Extreme Programming [Beck02], but is also used in other Agile methodologies and sometimes in sequential lifecycles. It helps developers focus on clearly-defined expected results. The tests are automated and are used in continuous integration.

Acceptance Test-Driven Development
Acceptance test-driven development [Adzic09] defines acceptance criteria and tests during the creation of user stories (see Section 1.2.2). Acceptance test-driven development is a collaborative approach that allows every stakeholder to understand how the software component has to behave and what the developers, testers, and business representatives need to ensure this behavior. The process of acceptance test-driven development is explained in Section 3.3.2.

Acceptance test-driven development creates reusable tests for regression testing. Specific tools support creation and execution of such tests, often within the continuous integration process. These tools can connect to data and service layers of the application, which allows tests to be executed at the system or acceptance level. Acceptance test-driven development allows quick resolution of defects and validation of feature behavior. It helps determine if the acceptance criteria are met for the feature.

Behavior-Driven Development
Behavior-driven development [Chelimsky10] allows a developer to focus on testing the code based on the expected behavior of the software. Because the tests are based on the exhibited behavior of the software, the tests are generally easier for other team members and stakeholders to understand.

Specific behavior-driven development frameworks can be used to define acceptance criteria based on the given/when/then format:

Given some initial context,
When an event occurs,
Then ensure some outcomes.

From these requirements, the behavior-driven development framework generates code that can be used by developers to create test cases. Behavior-driven development helps the developer collaborate with other stakeholders, including testers, to define accurate unit tests focused on business needs.

3.1.2 The Test Pyramid

A software system may be tested at different levels. Typical test levels are, from the base of the pyramid to the top, unit, integration, system, and acceptance (see [ISTQB_FL_SYL], Section 2.2). The test pyramid emphasizes having a large number of tests at the lower levels (bottom of the pyramid) and, as development moves to the upper levels, the number of tests decreases (top of the pyramid). Usually unit and integration level tests are automated and are created using API-based tools. At the system and acceptance levels, the automated tests are created using GUI-based tools. The test pyramid concept is based on the testing principle of early QA and testing (i.e., eliminating defects as early as possible in the lifecycle).

3.1.3 Testing Quadrants, Test Levels, and Testing Types

Testing quadrants, defined by Brian Marick [Crispin08], align the test levels with the appropriate test types in the Agile methodology. The testing quadrants model, and its variants, helps to ensure that all important test types and test levels are included in the development lifecycle. This model also provides a way to differentiate and describe the types of tests to all stakeholders, including developers, testers, and business representatives.

In the testing quadrants, tests can be business (user) or technology (developer) facing. Some tests support the work done by the Agile team and confirm software behavior. Other tests can verify the product. Tests can be fully manual, fully automated, a combination of manual and automated, or manual but supported by tools. The four quadrants are as follows:

- Quadrant Q1 is unit level, technology facing, and supports the developers. This quadrant contains unit tests. These tests should be automated and included in the continuous integration process.
- Quadrant Q2 is system level, business facing, and confirms product behavior. This quadrant contains functional tests, examples, story tests, user experience prototypes, and simulations. These tests check the acceptance criteria and can be manual or automated. They are often created during the user story development and thus improve the quality of the stories. They are useful when creating automated regression test suites.
- Quadrant Q3 is system or user acceptance level, business facing, and contains tests that critique the product, using realistic scenarios and data. This quadrant contains exploratory testing, scenarios, process flows, usability testing, user acceptance testing, alpha testing, and beta testing. These tests are often manual and are user-oriented.
- Quadrant Q4 is system or operational acceptance level, technology facing, and contains tests that critique the product. This quadrant contains performance, load, stress, and scalability tests, security tests, maintainability, memory management, compatibility and interoperability, data migration, infrastructure, and recovery testing. These tests are often automated.

During any given iteration, tests from any or all quadrants may be required. The testing quadrants apply to dynamic testing rather than static testing.

3.1.4 The Role of a Tester

Throughout this syllabus, general reference has been made to Agile methods and techniques, and the role of a tester within various Agile lifecycles. This subsection looks specifically at the role of a tester in a project following a Scrum lifecycle [Aalst13].

Teamwork
Teamwork is a fundamental principle in Agile development. Agile emphasizes the whole-team approach consisting of developers, testers, and business representatives working together. The following are organizational and behavioral best practices in Scrum teams:

- Cross-functional: Each team member brings a different set of skills to the team. The team works together on test strategy, test planning, test specification, test execution, test evaluation, and test results reporting.
- Self-organizing: The team may consist only of developers, but, as noted in Section 2.1.5, ideally there would be one or more testers.
- Co-located: Testers sit together with the developers and the product owner.
- Collaborative: Testers collaborate with their team members, other teams, the stakeholders, the product owner, and the Scrum Master.
- Empowered: Technical decisions regarding design and testing are made by the team as a whole (developers, testers, and Scrum Master), in collaboration with the product owner and other teams if needed.
- Committed: The tester is committed to question and evaluate the product's behavior and characteristics with respect to the expectations and needs of the customers and users.
- Transparent: Development and testing progress is visible on the Agile task board (see Section 2.2.1).
- Credible: The tester must ensure the credibility of the strategy for testing, its implementation, and execution, otherwise the stakeholders will not trust the test results. This is often done by providing information to the stakeholders about the testing process.
- Open to feedback: Feedback is an important aspect of being successful in any project, especially in Agile projects. Retrospectives allow teams to learn from successes and from failures.
- Resilient: Testing must be able to respond to change, like all other activities in Agile projects.

These best practices maximize the likelihood of successful testing in Scrum projects.

Sprint Zero
Sprint zero is the first iteration of the project where many preparation activities take place (see Section 1.2.5). The tester collaborates with the team on the following activities during this iteration:

- Identify the scope of the project (i.e., the product backlog)
- Create an initial system architecture and high-level prototypes
- Plan, acquire, and install needed tools (e.g., for test management, defect management, test automation, and continuous integration)
- Create an initial test strategy for all test levels, addressing (among other topics) test scope, technical risks, test types (see Section 3.1.3), and coverage goals
- Perform an initial quality risk analysis (see Section 3.2.1)
- Define test metrics to measure the test process, the progress of testing in the project, and product quality
- Specify the definition of "done"
- Create the task board (see Section 2.2.1)
- Define when to continue or stop testing before delivering the system to the customer

Sprint zero sets the direction for what testing needs to achieve and how testing needs to achieve it throughout the sprints.

Integration
In Agile projects, the objective is to deliver customer value on a continuous basis (preferably in every sprint). To enable this, the integration strategy should consider both design and testing. To enable a continuous testing strategy for the delivered functionality and characteristics, it is important to identify all dependencies between underlying functions and features.

Test Planning
Since testing is fully integrated into the Agile team, test planning should start during the release planning session and be updated during each sprint. Test planning for the release and each sprint should address the issues discussed in Section 1.2.5.

Sprint planning results in a set of tasks to put on the task board, where each task should have a length of one or two days of work. In addition, any testing issues should be tracked to keep a steady flow of testing.

Agile Testing Practices
Many practices may be useful for testers in a scrum team, some of which include:

- Pairing: Two team members (e.g., a tester and a developer, two testers, or a tester and a product owner) sit together at one workstation to perform a testing or other sprint task.
- Incremental test design: Test cases and charters are gradually built from user stories and other test bases, starting with simple tests and moving toward more complex ones.
- Ming mapping: Mind mapping is a useful tool when testing [Crispin08]. For example, testers can use mind mapping to identify which test sessions to perform, to show test strategies, and to describe test data.

These practices are in addition to other practices discussed in this syllabus and in Chapter 4 of the Foundation Level syllabus [ISTQB_FL_SYL].

3.2 Assessing Quality Risks and Estimating Test Effort

A typical objective of testing in all projects, Agile or traditional, is to reduce the risk of product quality problems to an acceptable level prior to release. Testers in Agile projects can use the same types of techniques used in traditional projects to identify quality risks (or product risks), assess the associated level of risk, estimate the effort required to reduce those risks sufficiently, and then mitigate those risks through test design, implementation, and execution. However, given the short iterations and rate of change in Agile projects, some adaptations of those techniques are required.

3.2.1 Assessing Quality Risks in Agile Projects

One of the many challenges in testing is the proper selection, allocation, and prioritization of test conditions. This includes determining the appropriate amount of effort to allocate in order to cover each condition with tests, and sequencing the resulting tests in a way that optimizes the effectiveness and efficiency of the testing work to be done. Risk identification, analysis, and risk mitigation strategies can be used by the testers in Agile teams to help determine an acceptable number of test cases to execute, although many interacting constraints and variables may require compromises.

Risk is the possibility of a negative or undesirable outcome or event. The level of risk is found by assessing the likelihood of occurrence of the risk and the impact of the risk. When the primary effect of the potential problem is on product quality, potential problems are referred to as quality risks or product risks. When the primary effect of the potential problem is on project success, potential problems are referred to as project risks or planning risks [Black07] [vanVeenendaal12].

In Agile projects, quality risk analysis takes place at two places.

- Release planning: business representatives who know the features in the release provide a high-level overview of the risks, and the whole team, including the tester(s), may assist in the risk identification and assessment.
- Iteration planning: the whole team identifies and assesses the quality risks.

Examples of quality risks for a system include:
- Incorrect calculations in reports (a functional risk related to accuracy)
- Slow response to user input (a non-functional risk related to efficiency and response time)
- Difficulty in understanding screens and fields (a non-functional risk related to usability and understandability)

As mentioned earlier, an iteration starts with iteration planning, which culminates in estimated tasks on a task board. These tasks can be prioritized in part based on the level of quality risk associated with them. Tasks associated with higher risks should start earlier and involve more testing effort. Tasks associated with lower risks should start later and involve less testing effort.

An example of how the quality risk analysis process in an Agile project may be carried out during iteration planning is outlined in the following steps:
1. Gather the Agile team members together, including the tester(s)
2. List all the backlog items for the current iteration (e.g., on a task board)
3. Identify the quality risks associated with each item, considering all relevant quality characteristics
4. Assess each identified risk, which includes two activities: categorizing the risk and determining its level of risk based on the impact and the likelihood of defects
5. Determine the extent of testing proportional to the level of risk.
6. Select the appropriate test technique(s) to mitigate each risk, based on the risk, the level of risk, and the relevant quality characteristic.

The tester then designs, implements, and executes tests to mitigate the risks. This includes the totality of features, behaviors, quality characteristics, and attributes that affect customer, user, and stakeholder satisfaction.

Throughout the project, the team should remain aware of additional information that may change the set of risks and/or the level of risk associated with known quality risks. Periodic adjustment of the quality risk analysis, which results in adjustments to the tests, should occur. Adjustments include identifying new risks, re-assessing the level of existing risks, and evaluating the effectiveness of risk mitigation activities.

Quality risks can also be mitigated before test execution starts. For example, if problems with the user stories are found during risk identification, the project team can thoroughly review user stories as a mitigating strategy.

3.2.2 Estimating Testing Effort Based on Content and Risk

During release planning, the Agile team estimates the effort required to complete the release. The estimate addresses the testing effort as well. A common estimation technique used in Agile projects is planning poker, a consensus-based technique. The product owner or customer reads a user story to the estimators. Each estimator has a deck of cards with values similar to the Fibonacci sequence (i.e., 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …) or any other progression of choice (e.g., shirt sizes ranging from extra-small to extra-extra-large). The values represent the number of story points, effort days, or other units in which the team estimates. The Fibonacci sequence is recommended because the numbers in the sequence reflect that uncertainty grows proportionally with the size of the story. A high estimate usually means that the story is not well understood or should be broken down into multiple smaller stories.

The estimators discuss the feature, and ask questions of the product owner as needed. Aspects such as development and testing effort, complexity of the story, and scope of testing play a role in the estimation. Therefore, it is advisable to include the risk level of a backlog item, in addition to the priority specified by the product owner, before the planning poker session is initiated. When the feature has been fully discussed, each estimator privately selects one card to represent his or her estimate. All cards are then revealed at the same time. If all estimators selected the same value, that becomes the estimate. If not, the estimators discuss the differences in estimates after which the poker round is repeated until agreement is reached, either by consensus or by applying rules (e.g., use the median, use the highest score) to limit the number of poker rounds. These discussions ensure a reliable estimate of the effort needed to complete product backlog items requested by the product owner and help improve collective knowledge of what has to be done [Cohn04].

3.3 Techniques in Agile Projects

Many of the test techniques and testing levels that apply to traditional projects can also be applied to Agile projects. However, for Agile projects, there are some specific considerations and variances in test techniques, terminologies, and documentation that should be considered.

3.3.1 Acceptance Criteria, Adequate Coverage, and Other Information for Testing

Agile projects outline initial requirements as user stories in a prioritized backlog at the start of the project. Initial requirements are short and usually follow a predefined format (see Section 1.2.2). Non-functional requirements, such as usability and performance, are also important and can be specified as unique user stories or connected to other functional user stories. Non-functional requirements may follow a predefined format or standard, such as [ISO25000], or an industry specific standard.

The user stories serve as an important test basis. Other possible test bases include:

- Experience from previous projects
- Existing functions, features, and quality characteristics of the system
- Code, architecture, and design
- User profiles (context, system configurations, and user behavior)
- Information on defects from existing and previous projects
- A categorization of defects in a defect taxonomy
- Applicable standards (e.g., [DO-178B] for avionics software)
- Quality risks (see Section 3.2.1)

During each iteration, developers create code which implements the functions and features described in the user stories, with the relevant quality characteristics, and this code is verified and validated via acceptance testing. To be testable, acceptance criteria should address the following topics where relevant [Wiegers13]:

- Functional behavior: The externally observable behavior with user actions as input operating under certain configurations.
- Quality characteristics: How the system performs the specified behavior. The characteristics may also be referred to as quality attributes or non-functional requirements. Common quality characteristics are performance, reliability, usability, etc.
- Scenarios (use cases): A sequence of actions between an external actor (often a user) and the system, in order to accomplish a specific goal or business task.
- Business rules: Activities that can only be performed in the system under certain conditions defined by outside procedures and constraints (e.g., the procedures used by an insurance company to handle insurance claims).
- External interfaces: Descriptions of the connections between the system to be developed and the outside world. External interfaces can be divided into different types (user interface, interface to other systems, etc.).

- Constraints: Any design and implementation constraint that will restrict the options for the developer. Devices with embedded software must often respect physical constraints such as size, weight, and interface connections.
- Data definitions: The customer may describe the format, data type, allowed values, and default values for a data item in the composition of a complex business data structure (e.g., the ZIP code in a U.S. mail address).

In addition to the user stories and their associated acceptance criteria, other information is relevant for the tester, including:

- How the system is supposed to work and be used
- The system interfaces that can be used/accessed to test the system
- Whether current tool support is sufficient
- Whether the tester has enough knowledge and skill to perform the necessary tests

Testers will often discover the need for additional information (e.g., code coverage) throughout the iterations and should work collaboratively with the rest of the Agile team members to obtain that information. Relevant information plays a part in determining whether a particular activity can be considered done. This concept of the definition of done is critical in Agile projects and applies in a number of different ways as discussed in the following sub-subsections.

Test Levels

Each test level has its own definition of done. The following list gives examples that may be relevant for the different test levels.

- Unit testing
  - 100% decision coverage where possible, with careful reviews of any infeasible paths
  - Static analysis performed on all code
  - No unresolved major defects (ranked based on priority and severity)
  - No known unacceptable technical debt remaining in the design and the code [Jones11]
  - All code, unit tests, and unit test results reviewed
  - All unit tests automated
  - Important characteristics are within agreed limits (e.g., performance)
- Integration testing
  - All functional requirements tested, including both positive and negative tests, with the number of tests based on size, complexity, and risks
  - All interfaces between units tested
  - All quality risks covered according to the agreed extent of testing
  - No unresolved major defects (prioritized according to risk and importance)
  - All defects found are reported
  - All regression tests automated, where possible, with all automated tests stored in a common repository
- System testing
  - End-to-end tests of user stories, features, and functions
  - All user personas covered
  - The most important quality characteristics of the system covered (e.g., performance, robustness, reliability)
  - Testing done in a production-like environment(s), including all hardware and software for all supported configurations, to the extent possible
  - All quality risks covered according to the agreed extent of testing
  - All regression tests automated, where possible, with all automated tests stored in a common repository
  - All defects found are reported and possibly fixed
  - No unresolved major defects (prioritized according to risk and importance)

User Story
The definition of done for user stories may be determined by the following criteria:

- The user stories selected for the iteration are complete, understood by the team, and have detailed, testable acceptance criteria
- All the elements of the user story are specified and reviewed, including the user story acceptance tests, have been completed
- Tasks necessary to implement and test the selected user stories have been identified and estimated by the team

Feature
The definition of done for features, which may span multiple user stories or epics, may include:

- All constituent user stories, with acceptance criteria, are defined and approved by the customer
- The design is complete, with no known technical debt
- The code is complete, with no known technical debt or unfinished refactoring
- Unit tests have been performed and have achieved the defined level of coverage
- Integration tests and system tests for the feature have been performed according to the defined coverage criteria
- No major defects remain to be corrected
- Feature documentation is complete, which may include release notes, user manuals, and on-line help functions

Iteration
The definition of done for the iteration may include the following:

- All features for the iteration are ready and individually tested according to the feature level criteria
- Any non-critical defects that cannot be fixed within the constraints of the iteration added to the product backlog and prioritized
- Integration of all features for the iteration completed and tested
- Documentation written, reviewed, and approved

At this point, the software is potentially releasable because the iteration has been successfully completed, but not all iterations result in a release.

Release
The definition of done for a release, which may span multiple iterations, may include the following areas:

- Coverage: All relevant test basis elements for all contents of the release have been covered by testing. The adequacy of the coverage is determined by what is new or changed, its complexity and size, and the associated risks of failure.
- Quality: The defect intensity (e.g., how many defects are found per day or per transaction), the defect density (e.g., the number of defects found compared to the number of user stories, effort, and/or quality attributes), estimated number of remaining defects are within acceptable limits, the consequences of unresolved and remaining defects (e.g., the severity and priority) are understood and acceptable, the residual level of risk associated with each identified quality risk is understood and acceptable.
- Time: If the pre-determined delivery date has been reached, the business considerations associated with releasing and not releasing need to be considered.
- Cost: The estimated lifecycle cost should be used to calculate the return on investment for the delivered system (i.e., the calculated development and maintenance cost should be considerably lower than the expected total sales of the product). The main part of the lifecycle

cost often comes from maintenance after the product has been released, due to the number of defects escaping to production.

3.3.2 Applying Acceptance Test-Driven Development

Acceptance test-driven development is a test-first approach. Test cases are created prior to implementing the user story. The test cases are created by the Agile team, including the developer, the tester, and the business representatives [Adzic09] and may be manual or automated. The first step is a specification workshop where the user story is analyzed, discussed, and written by developers, testers, and business representatives. Any incompleteness, ambiguities, or errors in the user story are fixed during this process.

The next step is to create the tests. This can be done by the team together or by the tester individually. In any case, an independent person such as a business representative validates the tests. The tests are examples that describe the specific characteristics of the user story. These examples will help the team implement the user story correctly. Since examples and tests are the same, these terms are often used interchangeably. The work starts with basic examples and open questions.

Typically, the first tests are the positive tests, confirming the correct behavior without exception or error conditions, comprising the sequence of activities executed if everything goes as expected. After the positive path tests are done, the team should write negative path tests and cover non-functional attributes as well (e.g., performance, usability). Tests are expressed in a way that every stakeholder is able to understand, containing sentences in natural language involving the necessary preconditions, if any, the inputs, and the related outputs.

The examples must cover all the characteristics of the user story and should not add to the story. This means that an example should not exist which describes an aspect of the user story not documented in the story itself. In addition, no two examples should describe the same characteristics of the user story.

3.3.3 Functional and Non-Functional Black Box Test Design

In Agile testing, many tests are created by testers concurrently with the developers' programming activities. Just as the developers are programming based on the user stories and acceptance criteria, so are the testers creating tests based on user stories and their acceptance criteria. (Some tests, such as exploratory tests and some other experience-based tests, are created later, during test execution, as explained in Section 3.3.4.) Testers can apply traditional black box test design techniques such as equivalence partitioning, boundary value analysis, decision tables, and state transition testing to create these tests. For example, boundary value analysis could be used to select test values when a customer is limited in the number of items they may select for purchase.

In many situations, non-functional requirements can be documented as user stories. Black box test design techniques (such as boundary value analysis) can also be used to create tests for non-functional quality characteristics. The user story might contain performance or reliability requirements. For example, a given execution cannot exceed a time limit or a number of operations may fail less than a certain number of times.

For more information about the use of black box test design techniques, see the Foundation Level syllabus [ISTQB_FL_SYL] and the Advanced Level Test Analyst syllabus [ISTQB_ALTA_SYL].

3.3.4 Exploratory Testing and Agile Testing

Exploratory testing is important in Agile projects due to the limited time available for test analysis and the limited details of the user stories. In order to achieve the best results, exploratory testing should be combined with other experience-based techniques as part of a reactive testing strategy, blended with

other testing strategies such as analytical risk-based testing, analytical requirements-based testing, model-based testing, and regression-averse testing. Test strategies and test strategy blending is discussed in the Foundation Level syllabus [ISTQB_FL_SYL].

In exploratory testing, test design and test execution occur at the same time, guided by a prepared test charter. A test charter provides the test conditions to cover during a time-boxed testing session. During exploratory testing, the results of the most recent tests guide the next test. The same white box and black box techniques can be used to design the tests as when performing pre-designed testing.

A test charter may include the following information:

- Actor: intended user of the system
- Purpose: the theme of the charter including what particular objective the actor wants to achieve, i.e., the test conditions
- Setup: what needs to be in place in order to start the test execution
- Priority: relative importance of this charter, based on the priority of the associated user story or the risk level
- Reference: specifications (e.g., user story), risks, or other information sources
- Data: whatever data is needed to carry out the charter
- Activities: a list of ideas of what the actor may want to do with the system (e.g., “Log on to the system as a super user”) and what would be interesting to test (both positive and negative tests)
- Oracle notes: how to evaluate the product to determine correct results (e.g., to capture what happens on the screen and compare to what is written in the user’s manual)
- Variations: alternative actions and evaluations to complement the ideas described under activities

To manage exploratory testing, a method called session-based test management can be used. A session is defined as an uninterrupted period of testing which could last from 60 to 120 minutes. Test sessions include the following:

- Survey session (to learn how it works)
- Analysis session (evaluation of the functionality or characteristics)

Deep coverage (corner cases, scenarios, interactions)The quality of the tests depends on the testers ability to ask relevant questions about what to test. Examples include the following:

- What is most important to find out about the system?
- In what way may the system fail?
- What happens if.....?
- What should happen when.....?
- Are customer needs, requirements, and expectations fulfilled?
- Is the system possible to install (and remove if necessary) in all supported upgrade paths?

During test execution, the tester uses creativity, intuition, cognition, and skill to find possible problems with the product. The tester also needs to have good knowledge and understanding of the software under test, the business domain, how the software is used, and how to determine when the system fails.

A set of heuristics can be applied when testing. A heuristic can guide the tester in how to perform the testing and to evaluate the results [Hendrickson]. Examples include:

- Boundaries
- CRUD (Create, Read, Update, Delete)
- Configuration variations
- Interruptions (e.g., log off, shut down, or reboot)

It is important for the tester to document the process as much as possible. Otherwise, it would be difficult to go back and see how a problem in the system was discovered. The following list provides examples of information that may be useful to document:

- Test coverage: what input data have been used, how much has been covered, and how much remains to be tested
- Evaluation notes*:* observations during testing, do the system and feature under test seem to be stable, were any defects found, what is planned as the next step according to the current observations, and any other list of ideas
- Risk/strategy list*:* which risks have been covered and which ones remain among the most important ones, will the initial strategy be followed, does it need any changes
- Issues, questions, and anomalies*:* any unexpected behavior, any questions regarding the efficiency of the approach, any concerns about the ideas/test attempts, test environment, test data, misunderstanding of the function, test script or the system under test
- Actual behavior: recording of actual behavior of the system that needs to be saved (e.g., video, screen captures, output data files)

The information logged should be captured and/or summarized into some form of status management tools (e.g., test management tools, task management tools, the task board), in a way that makes it easy for stakeholders to understand the current status for all testing that was performed.

3.4 Tools in Agile Projects

Tools described in the Foundation Level syllabus [ISTQB_FL_SYL] are relevant and used by testers on Agile teams. Not all tools are used the same way and some tools have more relevance for Agile projects than they have in traditional projects. For example, although the test management tools, requirements management tools, and incident management tools (defect tracking tools) can be used by Agile teams, some Agile teams opt for an all-inclusive tool (e.g., application lifecycle management or task management) that provides features relevant to Agile development, such as task boards, burndown charts, and user stories. Configuration management tools are important to testers in Agile teams due to the high number of automated tests at all levels and the need to store and manage the associated automated test artifacts.

In addition to the tools described in the Foundation Level syllabus [ISTQB_FL_SYL], testers on Agile projects may also utilize the tools described in the following subsections. These tools are used by the whole team to ensure team collaboration and information sharing, which are key to Agile practices.

3.4.1 Task Management and Tracking Tools

In some cases, Agile teams use physical story/task boards (e.g., whiteboard, corkboard) to manage and track user stories, tests, and other tasks throughout each sprint. Other teams will use application lifecycle management and task management software, including electronic task boards. These tools serve the following purposes:

- Record stories and their relevant development and test tasks, to ensure that nothing gets lost during a sprint
- Capture team members' estimates on their tasks and automatically calculate the effort required to implement a story, to support efficient iteration planning sessions
- Associate development tasks and test tasks with the same story, to provide a complete picture of the team's effort required to implement the story
- Aggregate developer and tester updates to the task status as they complete their work, automatically providing a current calculated snapshot of the status of each story, the iteration, and the overall release

- Provide a visual representation (via metrics, charts, and dashboards) of the current state of each user story, the iteration, and the release, allowing all stakeholders, including people on geographically distributed teams, to quickly check status
- Integrate with configuration management tools, which can allow automated recording of code check-ins and builds against tasks, and, in some cases, automated status updates for tasks

3.4.2 Communication and Information Sharing Tools

In addition to e-mail, documents, and verbal communication, Agile teams often use three additional types of tools to support communication and information sharing: wikis, instant messaging, and desktop sharing.

Wikis allow teams to build and share an online knowledge base on various aspects of the project, including the following:

- Product feature diagrams, feature discussions, prototype diagrams, photos of whiteboard discussions, and other information
- Tools and/or techniques for developing and testing found to be useful by other members of the team
- Metrics, charts, and dashboards on product status, which is especially useful when the wiki is integrated with other tools such as the build server and task management system, since the tool can update product status automatically
- Conversations between team members, similar to instant messaging and email, but in a way that is shared with everyone else on the team

Instant messaging, audio teleconferencing, and video chat tools provide the following benefits:

- Allow real time direct communication between team members, especially distributed teams
- Involve distributed teams in standup meetings
- Reduce telephone bills by use of voice-over-IP technology, removing cost constraints that could reduce team member communication in distributed settings

Desktop sharing and capturing tools provide the following benefits:

- In distributed teams, product demonstrations, code reviews, and even pairing can occur
- Capturing product demonstrations at the end of each iteration, which can be posted to the team's wiki

These tools should be used to complement and extend, not replace, face-to-face communication in Agile teams.

3.4.3 Software Build and Distribution Tools

As discussed earlier in this syllabus, daily build and deployment of software is a key practice in Agile teams. This requires the use of continuous integration tools and build distribution tools. The uses, benefits, and risks of these tools was described earlier in Section 1.2.4.

3.4.4 Configuration Management Tools

On Agile teams, configuration management tools may be used not only to store source code and automated tests, but manual tests and other test work products are often stored in the same repository as the product source code. This provides traceability between which versions of the software were tested with which particular versions of the tests, and allows for rapid change without losing historical information. The main types of version control systems include centralized source control systems and distributed version control systems. The team size, structure, location, and requirements to integrate with other tools will determine which version control system is right for a particular Agile project.

3.4.5 Test Design, Implementation, and Execution Tools

Some tools are useful to Agile testers at specific points in the software testing process. While most of these tools are not new or specific to Agile, they provide important capabilities given the rapid change of Agile projects.

- Test design tools: Use of tools such as mind maps have become more popular to quickly design and define tests for a new feature.
- Test case management tools: The type of test case management tools used in Agile may be part of the whole team's application lifecycle management or task management tool.
- Test data preparation and generation tools: Tools that generate data to populate an application's database are very beneficial when a lot of data and combinations of data are necessary to test the application. These tools can also help re-define the database structure as the product undergoes changes during an Agile project and refactor the scripts to generate the data. This allows quick updating of test data as changes occur. Some test data preparation tools use production data sources as a raw material and use scripts to remove or anonymize sensitive data. Other test data preparation tools can help with validating large data inputs or outputs.
- Test data load tools: After data has been generated for testing, it needs to be loaded into the application. Manual data entry is often time consuming and error prone, but data load tools are available to make the process reliable and efficient. In fact, many of the data generator tools include an integrated data load component. In other cases, bulk-loading using the database management systems is also possible.
- Automated test execution tools: There are test execution tools which are more aligned to Agile testing. Specific tools are available via both commercial and open source avenues to support test first approaches, such as behavior-driven development, test-driven development, and acceptance test-driven development. These tools allow testers and business staff to express the expected system behavior in tables or natural language using keywords.
- Exploratory test tools: Tools that capture and log activities performed on an application during an exploratory test session are beneficial to the tester and developer, as they record the actions taken. This is useful when a defect is found, as the actions taken before the failure occurred have been captured and can be used to report the defect to the developers. Logging steps performed in an exploratory test session may prove to be beneficial if the test is ultimately included in the automated regression test suite.

3.4.6 Cloud Computing and Virtualization Tools

Virtualization allows a single physical resource (server) to operate as many separate, smaller resources. When virtual machines or cloud instances are used, teams have a greater number of servers available to them for development and testing. This can help to avoid delays associated with waiting for physical servers. Provisioning a new server or restoring a server is more efficient with snapshot capabilities built into most virtualization tools. Some test management tools now utilize virtualization technologies to snapshot servers at the point when a fault is detected, allowing testers to share the snapshot with the developers investigating the fault.

4. References

4.1 Standards

- [DO-178B] RTCA/FAA DO-178B, Software Considerations in Airborne Systems and Equipment Certification, 1992.
- [ISO25000] ISO/IEC 25000:2005, Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE), 2005.

4.2 ISTQB Documents

- [ISTQB_ALTA_SYL] ISTQB Advanced Level Test Analyst Syllabus, Version 2012
- [ISTQB_ALTM_SYL] ISTQB Advanced Level Test Manager Syllabus, Version 2012
- [ISTQB_FA_OVIEW] ISTQB Foundation Level Agile Tester Overview, Version 1.0
- [ISTQB_FL_SYL] ISTQB Foundation Level Syllabus, Version 2011

4.3 Books

[Aalst13] Leo van der Aalst and Cecile Davis, "TMap NEXT® in Scrum," ICT-Books.com, 2013.

[Adzic09] Gojko Adzic, "Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing," Neuri Limited, 2009.

[Anderson13] David Anderson, "Kanban: Successful Evolutionary Change for Your Technology Business," Blue Hole Press, 2010.

[Beck02] Kent Beck, "Test-driven Development: By Example," Addison-Wesley Professional, 2002.

[Beck04] Kent Beck and Cynthia Andres, "Extreme Programming Explained: Embrace Change, 2e" Addison-Wesley Professional, 2004.

[Black07] Rex Black, "Pragmatic Software Testing," John Wiley and Sons, 2007.

[Black09] Rex Black, "Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing, 3e," Wiley, 2009.

[Chelimsky10] David Chelimsky et al, "The RSpec Book: Behavior Driven Development with Rspec, Cucumber, and Friends," Pragmatic Bookshelf, 2010.

[Cohn04] Mike Cohn, "User Stories Applied: For Agile Software Development," Addison-Wesley Professional, 2004.

[Crispin08] Lisa Crispin and Janet Gregory, "Agile Testing: A Practical Guide for Testers and Agile Teams," Addison-Wesley Professional, 2008.

[Goucher09] Adam Goucher and Tim Reilly, editors, "Beautiful Testing: Leading Professionals Reveal How They Improve Software," O'Reilly Media, 2009.

[Jeffries00] Ron Jeffries, Ann Anderson, and Chet Hendrickson, "Extreme Programming Installed," Addison-Wesley Professional, 2000.

[Jones11] Capers Jones and Olivier Bonsignour, "The Economics of Software Quality," Addison-Wesley Professional, 2011.

[Linz14] Tilo Linz, "Testing in Scrum: A Guide for Software Quality Assurance in the Agile World," Rocky Nook, 2014.

[Schwaber01] Ken Schwaber and Mike Beedle, "Agile Software Development with Scrum," Prentice Hall, 2001.

[vanVeenendaal12] Erik van Veenendaal, "The PRISMA approach", Uitgeverij Tutein Nolthenius, 2012.

[Wiegers13] Karl Wiegers and Joy Beatty, "Software Requirements, 3e," Microsoft Press, 2013.

4.4 Agile Terminology

Keywords which are found in the ISTQB Glossary are identified at the beginning of each chapter. For common Agile terms, we have relied on the following well-accepted Internet resources which provide definitions.

http://guide.Agilealliance.org/
http://whatis.techtargetcom/glossary
http://www.scrumalliance.org/

We encourage readers to check these sites if they find unfamiliar Agile-related terms in this document. These links were active at the time of release of this document.

4.5 Other References

The following references point to information available on the Internet and elsewhere. Even though these references were checked at the time of publication of this syllabus, the ISTQB cannot be held responsible if the references are not available anymore.

- [Agile Alliance Guide] Various contributors, http://guide.Agilealliance.org/.
- [Agilemanifesto] Various contributors, www.agilemanifesto.org.
- [Hendrickson]: Elisabeth Hendrickson, “Acceptance Test-driven Development,” testobsessed.com/2008/12/acceptance-test-driven-development-atdd-an-overview.
- [INVEST] Bill Wake, “INVEST in Good Stories, and SMART Tasks,” xp123.com/articles/invest-in-good-stories-and-smart-tasks.
- [Kubaczkowski] Greg Kubaczkowski and Rex Black, “Mission Made Possible,” www.rbcs-us.com/images/documents/Mission-Made-Possible.pdf.

5. Index